A Tunable Implementation of Quality-of-Service Classes for Dragonfly Networks

K A Brown*, S Chunduri*, R B Ross*, K Harms*, N McGlohon†, C D Carothers†, E Borch‡
*Argonne National Laboratory, †Rensselaer Polytechnic Institute, ‡Hewlett Packard Enterprise

***Abstract*—High-performance computer (HPC) networks are often shared by traffic from multiple applications with varying communication characteristics and resource requirements. These applications contend for shared buffers and channels, resulting in significant performance variations and slowdown of critical communication operations such as low-latency MPI collectives. In order to ensure predictable communication performance, network resources must be allocated based on the differing communication requirements of applications.**

Quality of Service (QoS) solutions using traffic classes can restrict the allocation of resources to improve performance predictability by defining traffic classes with specified resource allocations and assigning applications to these classes. However, it is impractical to define and enforce a different class allocation for each of the myriad of unique traffic patterns seen on HPC systems. This poses the challenge of maintaining a limited number of allocations to meet a wide range of application performance targets.

We propose a practical QoS implementation for large-scale, low-diameter networks, such as the dragonfly topology, using bandwidth shaping along with traffic prioritization to reduce the impact of interference on communication performance. Our design is applicable to an arbitrary number of traffic classes and can be tuned based on site-specific needs. We describe how a solution can be configured with 4 classes and deployed in production to match the varying application performance requirements of mixed workloads, providing performance predictability in a shared environment. The results show that our solution virtually eliminates the slowdown of high-priority traffic due to interference with lower-priority traffic, which also contributes to a significant reduction in run-to-run variability. We also demonstrate how port counters can be used to detect when a job-to-class assignment is inappropriate for a given system and when a workload is exceeding the rate limits of its class.

I. Introduction

The interconnection network of large supercomputers use high speed switches to route data across the network. Most HPC networks are shared by multiple applications with varying communication performance characteristics and bandwidth/latency requirements. The applications compete for link bandwidth and oversubscribe the link when the available bandwidth is less than the total bandwidth required by the competing application traffic flows.

Heavy traffic flows can unfairly monopolize the link bandwidth, particularly when each application is given equally unrestricted access to the network channels [1]. Without access restrictions, network contention becomes an issue that can result in reduced or delayed network access for different applications. This may unfairly harm the performance of certain types of application traffic. For example, this situation can lead to significant performance degradation in latency-sensitive communication traffic, such as small message MPI collectives, while potentially posing negligible impact on more latency-tolerant patterns such as checkpointing [2].

HPC networks use a variety of techniques to strike a balance between cost-efficient system use and application performance. Adaptive routing can be employed to improve application communication performance by re-routing packets around high-traffic areas of the network [3]. Adaptive routing can also be used to spread traffic more evenly across the network, ensuring more balanced use of network resources. Congestion management, aimed at diagnosing and treating network congestion when it occurs, works by temporarily reducing the rate at which packets are injected into the network and, over time, the total number of packets in-transit [4].

What routing and congestion management techniques don't provide is traffic separation and application isolation: allocation of network resources to specific applications or classes of traffic. Quality-of-service (QoS), on the other hand, can differentiate how resources are allocated to the traffic of different applications, enhancing application isolation and thus managing interference and resource contention [5]. With better management of resources and interference, application performance can be maintained despite significant network load.

System-level partitioning, giving applications their own exclusive sectors of the network system, is either resource-intensive and expensive, or it is impractical for networks that aim to balance traffic across all links with adaptive routing, such as the 1D dragonfly [3]. Additionally, physically partitioning the network can cause fragmentation which reduces overall system utilization. Resource-level partitioning using traffic classes to provide quality of service is a more efficient way of managing network resource without system fragmentation. Such a QoS solution differentiates between different types of traffic by placing them in different network-defined traffic classes.

Each traffic class is allocated separate network buffers and a guaranteed fraction of the channel bandwidth, based on the performance requirement of traffic assigned to the class [6]. Importantly, unused bandwidth of one class may be consumed by traffic of another class since this would not result in contention, providing flexible resource partitioning.

However, the effectiveness of traffic classes depends on how accurately the traffic assigned to the class matches the definition of each class. The OpenFabrics Interfaces Working Group [7] and several vendors [8] have defined traffic classes

that are expected to be used by HPC sites. Nevertheless, there is no universally ideal configuration for these classes due to the variations in workloads, interference patterns, and site-specific priorities across different HPC centers.

Studies up until now have focused on mainly priority-driven QoS or QoS based mainly on simple, course-grained bandwidth allocations [9]–[11]. Furthermore, there is limited knowledge available on how to precisely evaluate the suitability of a set QoS configurations for workloads with multiple distinct classes of traffic. HPC centers are running complex workloads that need more fine-grain resource allocations, hence, we also need better processes of fine-tuning QoS classes to match the requirements of the workload and site.

Our work aims to address these issues with the contributions as follows:

- We describe a practical method of implementing QoS classes on large-scale, low diameter networks to enable traffic differentiation, prioritization, and shaping using two rate limits per class.
- We propose a scheme for configuring and deploying QoS classes in production to match the varying application performance requirements of mixed workloads on production HPC systems.
- We evaluate the ability of our scheme to satisfy the relative performance goals of multiple workloads sharing a large-scale system.

Our solution reduces the impact of interference on critical communication operations, enabling latency-sensitive traffic to consistently achieve near baseline performance while ensuring bandwidth-intensive traffic is allocated the resources needed to achieve high bandwidth.

II. Background and Motivation

A. Communication Characteristics and Performance Targets

Application communication operations can be characterized as being either latency-bound or bandwidth-bound. Latency-bound transfers have low injection rates and the resulting communication time depends on individual packet latencies. These transfers are most sensitive to how fast individual packets progress through the network and are less sensitive to how many packets can be transferred in a given period. Small message MPI collectives such as MPI_Allreduce are examples of latency-bound operations that depend on low-latency packet transfers to meet performance targets. On the other hand, bandwidth-bound operations move relatively large amounts of data through the network and the overall communication time depends on the throughput instead of the latency of individual packets. That is, the communication time is most sensitive to how long it takes to receive all the data and less sensitive to the time it takes each packet to traverse the network. Bulk data I/O transfers are examples of bandwidth-bound operations.

To meet their respective performance targets, latency-bound traffic and bandwidth-bound traffic need access to the network resource in different manners. Keeping packet latencies low in latency-bounded flows require reducing the time spent queuing on the network by providing faster access to network channels. Bandwidth-bounded flows, however, require high injection rates or longer access to the shared channels in order to move a large amount of data through the network.

HPC facilities may further classify some traffic as having higher priority than others based on site-specific goals. For example, facility administrators may deem that certain mission-critical and latency-sensitive collective operations such as MPI_Allreduce should not be impeded and instead receive first priority access to network resources, regardless of source application. In contrast, they may decide that other types of traffic, such as network monitoring data, could be significantly delayed without adverse effects to the facility. The prioritization of different types of traffic may be based on, among other things, (i) loss in system throughput if a certain type of traffic is delayed (ii) or the impact on user experience when a certain type of traffic is slowed down.

B. Contention for Shared Network Channels

Contention for shared resources such as channel bandwidth causes interference to communication performance. Networks without QoS, however, treat all traffic indiscriminately, regardless of the traffic's purpose or source application. There are no restrictions or guarantees over the use of shared channels and buffers, and network resources are allocated to whichever application requests it first. As a result, competing traffic flows will be contending for network access. This contention will limit the amount of resources available to an application during periods of heavy loads when channels are oversubscribed. Interference also affects the order in which traffic flows get access to the channel, even when channels are not oversubscribed. With communication operations having unrestricted access to resources but differing requirements, some operations cannot acquire resources in the manner they need to consistently meet their performance targets. Conversely, some applications get access to more resources than they need to satisfy some user-defined performance goals. For example, the packets of small message MPI collectives are more sensitive to increased packet latencies than large I/O requests and therefore should be progressed faster through the network.

Interference over the network significantly degrade the performance of many HPC applications that are characterized by latency-sensitive collective communication patterns [12], [13]. The main cause of significant slowdown due to interference is increased queuing delays resulting from head-of-line (HoL) blocking, i.e when fast-draining messages get stuck behind slow-draining messages in shared buffers [10].

C. Adaptive Routing and Congestion Management

Interconnect congestion can be classified into two categories: *intermediate* and *endpoint* congestion [14]. Intermediate congestion occurs when multiple input ports on a router try to use the same output port, causing packets to get backed up in the buffers of the router. Endpoint congestion occurs due to application incasts – traffic from multiple source endpoints target the same destination endpoint, overwhelming the

endpoint's ability to accept all the incoming traffic. Adaptive routing can effectively address intermediate congestion by routing incoming packets to different output ports to avoid oversubscribing any single output port. However adaptive routing is ineffective against endpoint congestion because there are no alternative paths to the endpoint at the destination switch. With endpoint congestion, as traffic backs up from the target endpoint, adaptive routing will spread incoming traffic to less busy paths and potentially cause traffic to get backed up on those paths across the network as well. Congestion management schemes that appropriately abate the incast flows are essential for handling endpoint congestion. These solutions strive to curtail the injection volume at the congestion-causing sources based on how many packets can be consumed at the endpoints [4].

D. QoS Considerations

QoS mechanisms are not designed to address endpoint or intermediate congestion. QoS is used to decide which packet to send based on priority and bandwidth specifications. This is orthogonal to both adaptive routing, which determines the path a packet should take, and congestion management, which determines if a packet should be injected into the network based on the state of congestion in the network.

Traffic flows can be regulated using QoS to ensure that flows acquire the resources they require to meet their performance targets [9]–[11]. As the available bandwidth and packet latencies affect an application's ability to meet its relative performance targets, QoS mechanisms should allow for regulating resources that affect the resulting packet latency and bandwidth allocated to traffic flows.

Packet latencies are mostly affected by high queuing delays and head-of-line (HoL) blocking on routers. To prevent high-packet latencies in latency-sensitive flows, queuing delays can be reduced by giving the flow a higher arbitration priority for the output port than other flows. This will allow the packet to be injected ahead of lower priority packets. Additionally, to prevent HoL blocking, different flows should use different virtual channel buffers. Placing latency-sensitive communication in a separate class that uses separate buffering and a high priority has been shown to reduce the latency, similar to the expedited forwarding per-hop behavior (PHB) of TCP differentiated services [5].

Bandwidth guarantees (or assured injection rates) can be defined on each class to ensure a minimum amount of bandwidth is available to traffic using that class when channels are oversubscribed. In addition, peak bandwidth (or peak injection rate) limits can be set for each class to limit the amount of bandwidth a particular class can use at its given priority level. Both limits, if set properly, allow different classes to receive enough bandwidth to make forward progress and not be starved by higher priority classes, as shown in Section IV. This prevents any class from unintentionally dominating the bandwidth at the expense of all other classes.

Ideally, we would like to isolate the application traffic from from each other to ensure that each flow is allocated the resources it needs to meet its performance targets. This is impractical because assigning resources to each application based on the application's specific requirements requires a vast amount of hardware resources. Therefore, a more realistic solution is to group applications with similar characteristics and performance targets in the same class and allocate resources on a per-class basis. This is the prevailing solution for most practical QoS solutions [6], [9], [11].

III. Design of a Tunable QoS Solution

A. Traffic Prioritization and Shaping

We propose a QoS mechanism that supports the diverse set of workloads expected on modern HPC systems. This solution allows for configuring an arbitrary number of traffic classes with independent virtual channel buffers and unique relative priorities to enable traffic prioritization. To achieve bandwidth shaping, each class is configurable with two rate limits: an *assured rate (AR) limit* and a *peak rate (PR) limit*, where AR $\leq$ PR. This solution is based on the Two Rate Three Color Marking design [15]. The packets that are ready to be sent from each class are marked as either green, yellow, or red, depending on the current injection rate of its respective class. A packet is marked as red if the class exceeds its PR, or it is marked as yellow if only the AR has been exceeded. And, a packet is marked as green if its class does not exceed its AR. Marking is done each injection cycle for the purpose of output port arbitration, and stalled packets are re-marked in subsequent cycles. The packet content is unchanged and marking information is not communicated downstream.

Arbitration is priority-based within the constraints of the rate limits. That is, green packets are sent first from higher priority classes. If there are no green packets that can be sent, then yellow packets are sent in a similar priority order. If neither green nor yellow packets can be sent, a red packet will be chosen from a class by round-robin – priorities are ignored and each class has an equal chance of getting access to the output port. Note that flow control can stop any class from sending if downstream buffers are unavailable, in which case a packet form another class is sent.

A separate token bucket is used to meter each of the two rate limits per class. Tokens accumulate in a bucket at the rate of the limit it meters, i.e., the assured rate bucket will accumulate tokens at the assured rate limit defined on the class, etc. Whenever a green or yellow packet is sent from a class, a token removed from each of the two buckets with available tokens. No token can be removed when a red packets is sent because the peak rate limit has been exceed, at which point both buckets are empty.

Fig. 1 illustrates our solution being used for two QoS classes with traffic from three applications.

B. Assigning Traffic To Classes

We propose grouping applications based on the similarity of their priorities and performance characteristics. Classes have strict priorities relative to each other, so we first consider the traffic flow's priority relative to that of other flows.

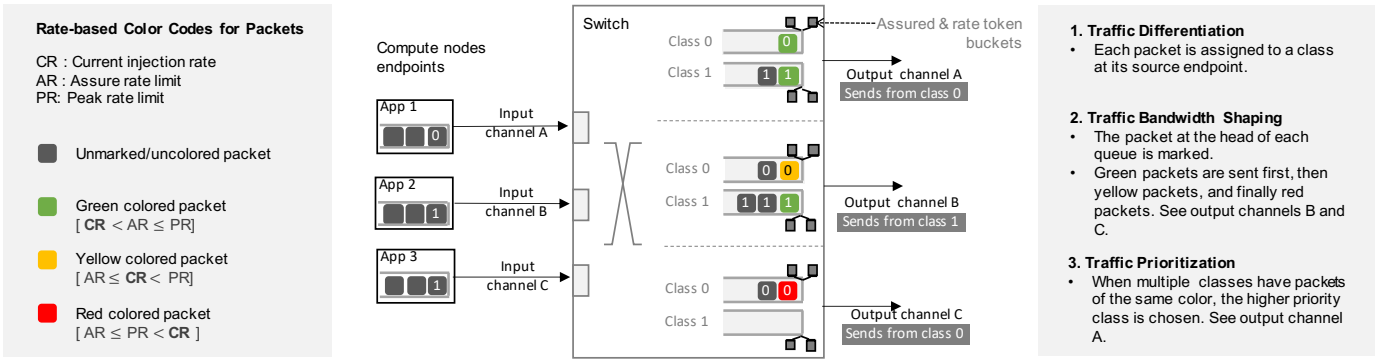

Fig. 1. Illustration of our QoS solution design. Packets are assigned to classes at compute node endpoints, placed in the appropriate class buffers on switches, and then are colored and compete for access to output channels based on the QoS policy. Both App 1 and 2 assign their traffic to class 1.

When all other factors are equal, the priority will determine which flow progresses first and achieve lower latency. Placing flows with similar communication requirements, rather than adversarial patterns, in the same class can still yield improved performance. We propose this grouping based on the following traffic patterns:

- Latency-sensitive traffic: traffic that is primarily latency-bound, does not require high bandwidth, and is needed for important, small-to-medium sized collective operations.
- Bandwidth-sensitive traffic: traffic that moves a lot of data at once and requires high bandwidth. Individual packet latency is not important since higher packet latencies are amortized by the cumulative transmission delay of all the packets in the data being transferred.
- Optional traffic: traffic than can be ignored without significant impact to overall productivity and user experience, such as periodically scraping network counters.
- Other: Traffic with mixed bandwidth and latency sensitivity.

We argue that these groups form a good starting point for categorizing most HPC traffic and are similar to industry standard traffic classification [7]. However, other grouping strategies may be used to match the needs of the user with the requirement that traffic sharing the same group are not adversarial to each other in terms of latency and bandwidth sensitivity. We can then use the following QoS class definitions to regulate the collective flow of traffic per group:

1) Low latency: Guarantees low-latency for traffic in the latency-sensitive group. This class should have the highest arbitration priority to reduce queuing delays and lower rate limits to prevent starvation of lower-priority classes. Lower rate limits also help to guarantee low packet latencies by limiting the competition for resources within the class.

2) Best effort: Makes best effort progress of traffic with mixed latency-sensitivity and bandwidth requirements. This class should be used by most application traffic except for low-latency traffic, bulk data I/O, and traffic assigned to other classes. This class should be allocated with sufficiently high bandwidth guarantees based on the high volume of data transferred by its expected workload.

3) Bulk data: Used by traffic with high bandwidth requirements and negligible latency sensitivity. This class should be allocated bandwidth commensurate with the I/O throughput of the system and the importance of I/O performance to the system workloads.

4) Scavenger: Guarantees only a small fraction of the system bandwidth to ensure progress of non-essential traffic, such as performance data etc., in the background when the channel is free. This class should have the lowest priority and low rate limits to prevent it from impacting the performance of higher priority jobs in the other classes.

Our QoS implementation supports these and other class definitions by tuning the assured and peak rate limits on each class for flexible bandwidth shaping and adjusting the relative arbitration priority to either guarantee low-latency or restrict interference. The following sections demonstrate how our implementation uses shaping and prioritization of traffic to deliver consistent application performance and meet performance targets.

IV. Evaluation of QoS Solution

A. CODES Simulation Toolkit

To collect the data evaluated in this work, we use the CODES HPC interconnection network simulator [16]. CODES is a Parallel Discrete Event Simulation (PDES) toolkit built on top of the Rensselaer Optimistic Simulation System (ROSS) [17] PDES engine. CODES allows for fine-grained, link-level simulations of packets moving across high-performance networks. Additionally, these simulations allow for testing and evaluation of different technologies such as adaptive routing algorithms, congestion management, and, as demonstrated in this work, QoS techniques. We implemented our solution design in CODES as outlined in the previous section.

B. Network Setup

We simulate a 8320-node system connected using a 1D dragonfly network. The network consists of 1040 routers with 16 routers per group. Each router has 8 terminal channels, 15 local channels, and 4 global channels. The ratio of terminal channels to global channels results in a 2:1 taper of the

network bandwidth, similar to the Theta, Edison, Malbec, and Shandy systems [18], which increases the potential for contention among competing traffic flows. We use 25 GB/s injection bandwidth for all channels, 10 ns delay for terminal and local channels, and 100 ns delay for global channels. The simulated router delay is 300 ns and the network packet size is set to 160 bytes. We use a progressive-adaptive routing algorithm for the network and random node allocation to assign nodes to jobs. Studies show that this node allocation strategy improves application throughput for dragonfly systems [19].

We simulate 4 active QoS classes with 4 different priorities.

1) Class 0: has the highest priority and is earmarked as the **low-latency class**. Latency sensitive traffic, such as small message MPI collectives, that will use this class should have low injection loads, so we set relatively low rate limits on this class.

2) Class 1: has the second highest priority and is earmarked as the **best effort class**. Most application traffic will use this class. It is given the highest bandwidth allocation of all classes.

3) Class 2: is earmarked as the **bulk data class** to carry bulk I/O data across the system. The aggregate I/O load on the system will be limited, in part, by the number of I/O nodes in the system. Therefore, we allocate bandwidth to this class relative to the fraction of system occupied by I/O nodes. This would ensure progression of I/O traffic.

4) Class 3: has the lowest priority and is earmarked as the **scavenger class**. We use this class to demonstrate how very low-priority traffic can be facilitated on the network without impacting the performance of other traffic by giving this class a low fraction of the system bandwidth. Setting the assured limit to a value greater than zero ensures forward progress for traffic in this class. However, if forward progress guarantees are not required, the assured bandwidth limit can be set to zero and the traffic will potentially be starved indefinitely by higher-priority flows.

The values chosen for the per-class bandwidth allocations are for demonstrating the capabilities of this QoS solution. Each site would configure the classes to match the composition of their site-specific workloads and administration policies, such as high-priority applications/users.

C. Workload Setup

To generate interference on the network, we use multiple jobs generating uniform random traffic with every message being sent to a random destination rank within the same job. We use 640 B messages and vary the injection load of each job by varying the delay between injecting successive messages. These small messages allow us to avoid local incasts and evenly spread load across the system. Other synthetic patterns such as random-permutation cause congestion hotspots for which QoS is not the appropriate solution. Such patterns require a congestion management solution, which is outside the scope of this work.

We use a Scalable Workload Model (SWM) [20] of MPI_Allreduce, a common operation on HPC systems [21], to simulate latency sensitive traffic. SWMs are skeletons of applications and benchmarks that capture the communication patterns of the workload that they model. Each allreduce SWM benchmark performs multiple calls to MPI_Allreduce, reducing 8 bytes of data across all ranks of the job.

TABLE I
QoS Class configuration for single-port study

Class	Priority	Assured rate limit (%)	Peak rate limit (%)
0	P0 (highest)	20	100
1	P1	45	100
2	P2	35	100
3	P3 (Lowest)	0	0

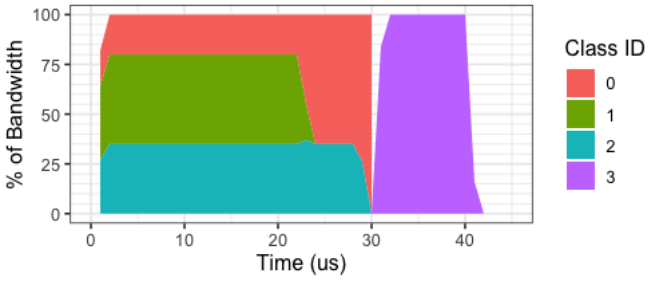

Fig. 2. QoS bandwidth partition over a single port. Each class is carrying a stream of 1000 packets.

D. Single-port Traffic Shaping

To demonstrate the bandwidth shaping capabilities, we simulate four identical traffic flows sharing the bandwidth of a single channel. The port is configured with four traffic classes, with each class being used by a single flow. The class definitions are provided in Table I. Each flow streams 1000 packets over a shared router-to-router channel. The stacked area plot in Fig. 2 shows the results as each class attempts to use 100% of the channel bandwidth until they complete sending their respective payloads.

The results in Fig. 2 illustrates how our implementation shapes traffic bandwidth through a single port. At the start of the simulation, classes 0, 1, and 2 are able to share the port's bandwidth at their respective assured rates. Class 3 is starved and unable to send because it is not assured a fraction of the bandwidth and the port is fully utilized by traffic from the other classes. After traffic from classes 1 and 2 complete, class 0 increases its utilization up until its peak rate. With Classes 0, 1, and 2 having a peak rate of 100%, Class 3 can use the port only after the other classes have completed sending.

E. System-wide Traffic Shaping

After evaluating bandwidth shaping on a single port, we assess the impact of system-wide bandwidth shaping using QoS and four identical synthetic traffic jobs generating uniform random traffic. Each rank of each job injects 10,000 640 B messages at the peak channel injection rate, with each message being sent to a randomly chosen remote rank of the same job. With our 2:1 tapered network, the total offered load is twice the global bandwidth. Each job is assigned to a different class

as shown in Table II. The classes' peak rates are set to equal their respective assured rates to show strict traffic shaping.

TABLE II
QOS CLASS CONFIGURATION FOR SYSTEM-WIDE STUDY

Job	Class	Priority	Assured rate limit (%)	Peak rate limit (%)
0	0	P0 (highest)	10	10
1	1	P1	60	60
2	2	P2	20	20
4	3	P3 (lowest)	10	10

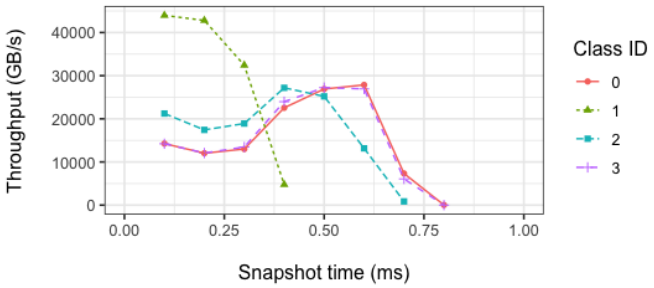

Fig. 3. Change in class throughput over time based on interference and QoS bandwidth allocation. Each class is carrying the same total volume of traffic.

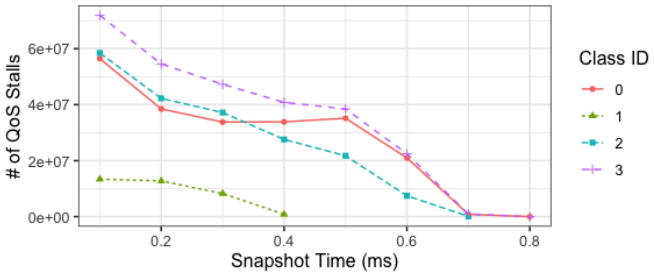

Fig. 4. System-wide QoS stalls per class during each snapshot window of the bandwidth shaping study.

Throughput results for this study are shown in Fig. 3. Class 1 is able to achieve the highest throughput since it had the largest bandwidth allocation. Class 0 and 3 achieve similar throughput because their traffic loads and bandwidth allocations are equal. The slight variation in their performance is attributed to their different random node assignments and different priorities.

A class is blocked from injecting its next packet if any one of three conditions are met. We describe these cases as the following types of *QoS stalls*:

Green stall: The class has a green packet and a higher priority class also has a green packet ready to inject.

Yellow stall: The class has a yellow packet and either a higher-priority class has a yellow packet ready to send or any other class has a green packet ready to send.

Red stall: The class has a red packet and either any other class has a green/yellow packet ready to send or it loses to another class in round-robin arbitration.

Fig. 4 shows the total number of QoS stalls per class and Fig. 5 provides a breakdown of the types of stalls. Green packets are stalled only when a higher priority class has green packets. For this reason, when two classes have equal bandwidth allocations, as the case with classes 0 and 3, the class with lower priority will experience more frequent green stalls. Class 0 and 3 have equal shares of the bandwidth but class 3 gets stalled because of lack of priority while class 0 does not since it has the highest priority of all classes.

Yellow stalls occur only when a class exceeds its assured rate and is still within its peak rate. There were no yellow stalls in this scenario because the assured rates and peak rates were equal for each class.

Red stalls occur when the traffic has exceeded the peak rate limit of its class. Class 0 experiences many red stalls because job 0 occupies 25% of the system and has an aggregate injection rate of 50% of the global bandwidth on this tapered system. This exceeds the 10% peak rate limit of class 0.

As discussed in Section III, the class with the highest priority should be configured as the low-latency class that will have priority arbitration. However, Fig 6 reports that packets in class 0 experience high tail latencies due to the high number of QoS red stalls experienced by class 0. This highlights the issue of having high-bandwidth flows in a low-latency class.

F. Low-latency Assurance and Repeatability

To demonstrate the ability of the QoS implementation to guarantee low-latency and reduced performance variability for high-priority latency-sensitive traffic, we evaluate a system running multiple All_reduce jobs along with multiple UR jobs generating background traffic at different rates. Table III shows the workload configuration on our simulated 8320-node tapered 1-D dragonfly system and the assigned class rate limits are shown in Table IV.

Each allreduce job calls 10 MPI_Allreduce operations to reduce 8 bytes of data. We run four instances of allreduce on 32 nodes and two instances of allreduce on 256 nodes. The rest of the system is occupied with three UR traffic patterns. The allreduce jobs are smaller than the UR jobs because small jobs are potentially more sensitive to interference from large jobs. The multiple MPI_Allreduce calls in each job and the multiple instances of jobs at different scales show how effectively QoS can facilitate performance repeatability. The injection rates are 80%, 60%, and 20% for UR-1, UR-2, and UR-3, respectively. The rates were chosen to (i) align with the peak rate limit of the class assigned to the job and (ii) approximate the expected load that would be seen by that class. The UR jobs inject traffic at a steady rate until every allreduce job completes, then the UR jobs stop injecting data. After this point, in-transit UR packets drain from the network.

We compare three configurations to evaluate the effects of QoS: (i) *Standalone*, (ii) interference with QoS *(QoS)*, and (iii) interference without QoS *(no-QoS)*. We measure the performance of Standalone allreduce jobs by running the jobs on an idle system without background traffic. For the interference without QoS, we run the allreduce jobs at the

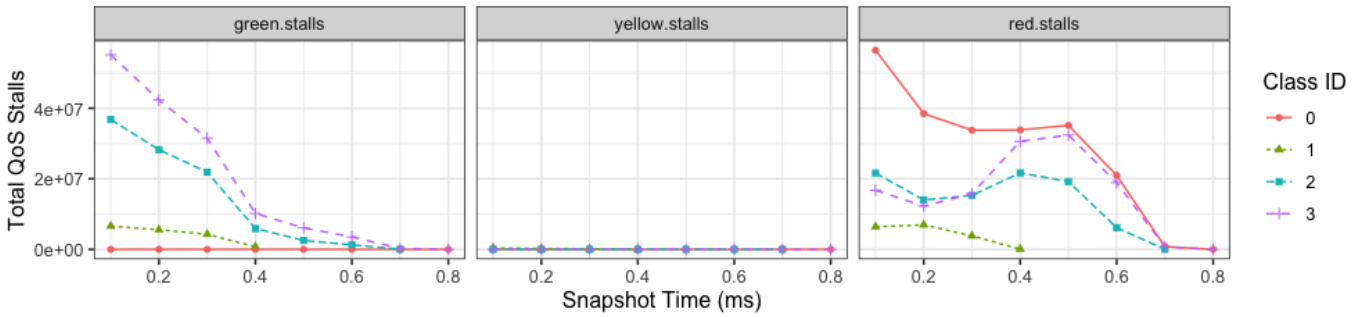

Fig. 5. Breakdown of System-wide QoS stalls per class during each snapshot window of the bandwidth shaping study.

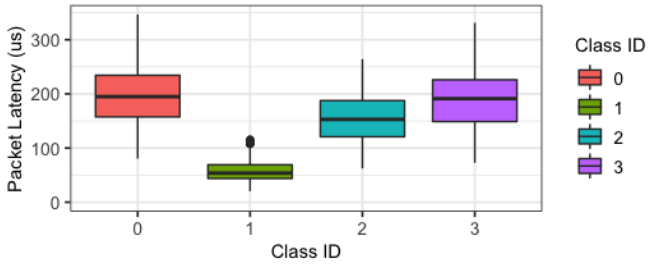

Fig. 6. Distribution maximum packet latencies seen by nodes using each class.

TABLE III
WORKLOAD CONFIGURATION FOR LOW-LATENCY STUDY

Job_id	#_Nodes	App	Class
0	32	allreduce32	0
1	32	allreduce32	0
2	32	allreduce32	0
3	32	allreduce32	0
4	256	allreduce256	0
5	256	allreduce256	0
6	4160	UR-1	1
7	1760	UR-2	2
8	1760	UR-3	3

same time as the UR jobs without using QoS, i.e., traffic from all jobs share a single class. For interference runs with QoS, each job uses their assigned class. The same node allocations are used for all three configurations. All jobs begin once the simulation starts. We do not explicitly specify a warm-up period because of the volume of traffic being generated; the iterations of MPI_Allreduce calls within each allreduce job and the duration of the runs sufficiently amortizes the time it takes for the network to warm up. We will show the network

TABLE IV
QOS CLASS CONFIGURATION FOR LOW-LATENCY STUDY

Class	Assured rate (%)	Peak rate (%)
0	5	10
1	30	80
2	20	60
3	5	20

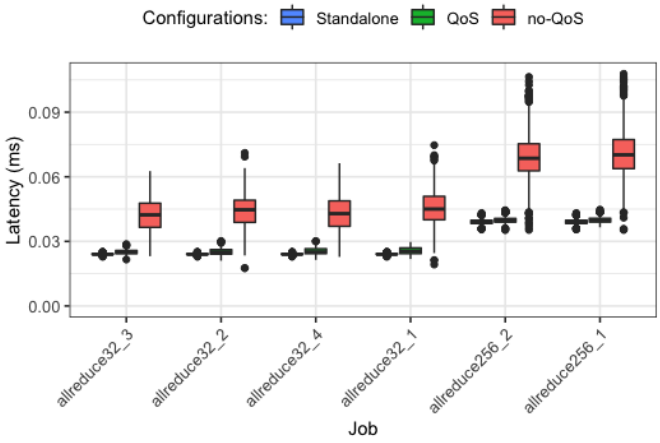

Fig. 7. Distribution of All_Reduce operation latency across job ranks.

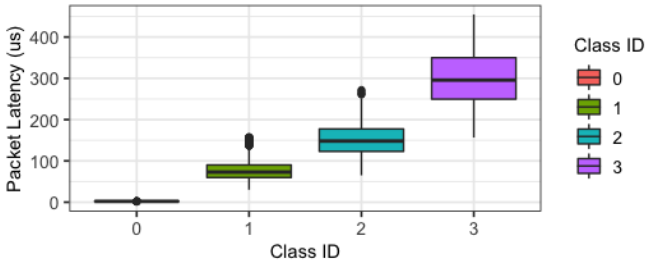

Fig. 8. Distribution maximum packet latencies seen by nodes using each class.

performance over several time steps (or snapshot periods) to confirm this.

Fig. 7 shows the distribution of MPI_Allreduce latency across all ranks of a job for different system configurations. The first boxplot for each job is the performance with the *Standalone* configuration while the middle boxplot shows performance when background traffic is causing interference and each job is using their assigned class, i.e., *QoS*. The third boxplot for each jobs shows the MPI_Allreduce latency when it is not isolated from the background traffic, i.e. *no-QoS*. We achieve near *Standalone* performance with the *QoS* configuration, where allreduce jobs use the low-latency class

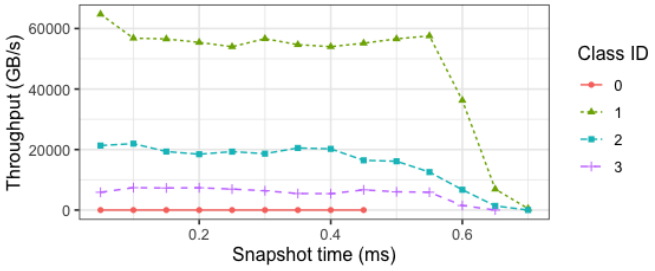

Fig. 9. Change in class throughput over time based on interference and QoS bandwidth allocation. The injection rate into each class is different and injection ends for all classes when the job in class 0 terminates.

and background traffic is assigned to different classes. The *QoS* config. eliminates the increase in maximum latency due to interference that is seen in the *no-QoS* config. Similarly, the maximum packet lantencies seen by all nodes in class 0 with the *QoS* config. are significantly lower than those of other classes, as illustrated in Fig. 8.

Fig. 9 shows the per-class throughput results for the *QoS* config. The load in class 0 is over 2 orders of magnitudes less than that of the all other classes and is within the 5% assured rate limit of class 0. Fig. 10 shows the breakdown of system-wide QoS stalls for this low-latency experiment. Class 0 experienced negligible stalls because of its low load and high priority. Additionally, there were negligible red stalls in this experiment because each class's peak rate limit was equal to or greater than the load of the traffic assigned to that class. Unlike in the traffic shaping experiments in the previous subsections, this peak rate limit is higher than the assured rate limit on each class, causing many yellow stalls. Setting different assured and peak rate limits offers more flexibility in bandwidth shaping since consumption of non-assured bandwidth can be driven by priority instead of by a round-robin process.

We increase the injection rate of UR-2 and UR-3 to 80%, thereby increasing the intensity of interference from background traffic. For this configuration, the injection rates of traffic in classes 2 and 3 exceed the peak rate limit of the class and show the results in Fig. 11. These results confirm that we are still able to achieve near Standalone performance consistently for each allreduce job and for the multiple MPI_Allreduce calls within each job, indicated by the short whiskers and positions of the outliers. The load increase in the two lowest priority classes, 2 and 3, had a negligible impact on the performance of traffic in classes 0 and 1. In addition to the allreduce jobs having consistently near-ideal performance, the traffic in class 1 was able to maintain high throughput due to the bandwidth allocation guaranteed by QoS, as illustrated in Fig. 12.

V. Discussion

A. Traffic Load and QoS Rate Limits

QoS enforces defined resource allocations to different classes in an effort to manage the competition for resources and reduce slowdown due to interference. The allocation of resources should be defined in a manner to ensure each flow can progress at an acceptable pace without any unintended impact to other flows. We demonstrated that setting an assured rate limit for a class will ensure the class is not starved, regardless of the intensity of interference from other classes.

The proper configuration of the assured and peak rate limits for a class depends on the performance targets of its expected workloads, as outlined in Section II-D. We demonstrate that equating the peak rate of the class to the peak expected traffic load is a good starting point to finding a suitable value. For the assured rate limits, we recommend starting with the minimum required rate needed to attain acceptable throughput and/or latency for traffic assigned to the class when the channels are oversubscribed. These limits can then be tuned using the QoS stall metrics as guides. The number of yellow and red stalls are indicators of constraints due to rate limits since these packets get stalled only when the assured/peak rate has been exceeded. Additionally, the number of green stalls is an indicator of constraints due to insufficient priority since packets are marked green when the class has not exceed its assured rate. The number of acceptable stalls will depend on the workload and the desired traffic shaping outcome.

For QoS to work as expected, it is important that workloads are assigned to their appropriate classes. For example, a properly tuned low-latency class will not be able to guarantee low packet latencies if a bandwidth-intensive flow that exceeds the peak rate is assigned to that class. This is shown in Fig. 6, where class 0 packet latencies are very high due to the high load in that class. The QoS stall counters can be used here to flag an inappropriate traffic-to-class assignment. An unexpectedly high number of red stalls in a class could indicate the traffic has exceeded the class' expected load.

B. Production Deployment

Evidence indicates that users are unlikely to utilize advanced system capabilities effectively if it requires detailed knowledge of the service [22]. QoS solutions are not exceptions. Wide users adoption of QoS require user-friendly QoS interfaces. A simple way of exposing QoS mechanisms to the users is to automatically assign different workloads to specific classes, such as by having the MPI library automatically assign certain operations to specific classes or having the I/O subsystem assign all I/O traffic to the bulk data class. This would be transparent to the user while providing the full benefits of QoS. Another benefit of this solution is that administrators can define system-wide configurations to restrict what type of traffic can be assigned to which class, preventing inappropriate traffic-to-class assignment.

VI. Related Work

Several QoS solutions have been proposed for low-diameter networks such as dragonfly and fat-tree networks. Most of these vary the arbitration priority in some manner to reduce packet latencies, or regulate the bandwidth allocation to different flows, or both [9], [10], [11]. These studies discuss *service levels* instead of *QoS classes* depending on the architecture

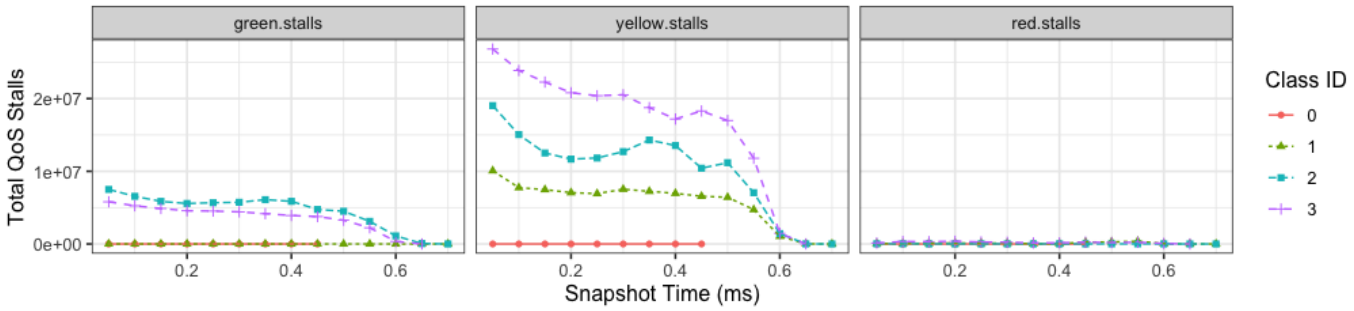

Fig. 10. Breakdown of System-wide QoS stalls per class during each snapshot window of the low latency study. The injection of traffic in classes 1, 2, and 3 are equal to the class's peak rate limit shown in Table IV.

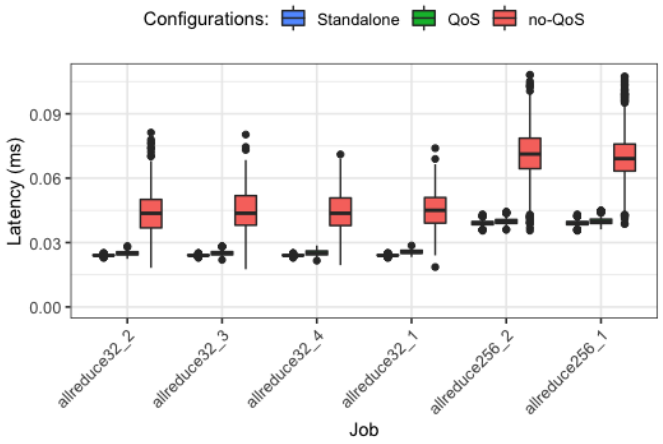

Fig. 11. Distribution of MPI_AllReduce operation latency across ranks of each job. Background traffic is being injected at 80% of the peak rate.

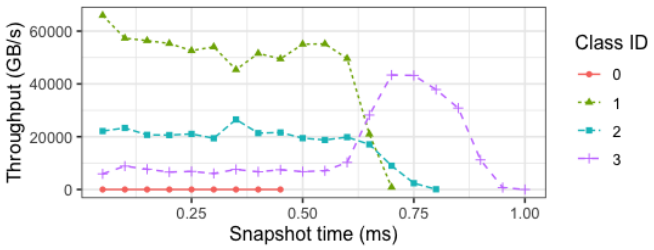

Fig. 12. Change in class throughput over time based on interference and QoS bandwidth allocation. The injection rate into classes 1, 2, and 3 is 80% of maximum link rate and all traffic injections end when the job in class 0 terminates.

used in their respective study. For consistency, our reference to *QoS classes* include *service levels* since the concepts are the same.

Wilke and Kenny [10] propose using four different traffic classes to cater for a wide range of HPC workloads. Their solution uses two low-latency classes for small, latency-sensitive communication and two high-throughput classes for bandwidth-intensive traffic. One of the main contributions of their work is to reduce the number of virtual channel buffers required to support four classes by using minimal routing for the low-latency classes. The classes with minimal routing do not need the additional buffers required for deadlock-free adaptive routing. This limits the flexibility of their solution since two classes can only support low-latency traffic because of the classes' minimal routing constraint. Our solution supports an arbitrary number of classes, with each class being fully configurable to support any type of traffic. While [10] studies an interesting concept to reduce the buffer requirements of QoS, we have not seen any indications that the reduction in buffer size requirements warrant the loss in flexibility. Another notable distinction is that their work splits the bandwidth equally across all four classes, while our implementation supports classes with bandwidth limits that can be tuned to better match the expected workloads.

Multiple studies [6], [9], [11] proposed grouping application traffic flows into separate QoS classes. Savoie et al. [11] used only priority as constraints for the QoS classes. These solutions are insufficient for having fine-grained control over bandwidth allocations as high priority classes can consume most of the bandwidth and unfairly starve lower-priority classes for long periods.

Savoie et al. [23] proved that assigning entire application to classes was not very effective at improving communication performance. Along with Savoie et al., other studies [9], [11] demonstrate that selectively giving priority to MPI collectives can better improved overall application performance. They show that grouping traffic flows with similar bandwidth intensities can reduce the interference caused by high-utilization (or bandwidth-intensive) traffic on low-utilization traffic. Our solution supports this type of assignment where different traffic flows within an application can be placed in different classes. We go further to propose using groups that match not only the application utilization, but also the performance targets of the traffic. This allows system administrators to differentiate between types of traffic that have similar utilization but different importance to the site.

Jakanaovic et al. [6] recommends allocating most of the bandwidth to the class with low bandwidth requirement. This has the potential to degrade overall system performance and invalidates the benefits of QoS if a bandwidth intensive

workload is assigned to this class, as they also noted in their work. The flexibility of our dual rate solution with class prioritization allows latency-sensitive traffic to be guaranteed good performance without requiring assured bandwidth allocations and allows the bandwidth to be allocated appropriately to other classes.

VII. Conclusions

HPC networks often run multiple applications with differing communication patterns. These patterns compete for network resources, and this competition causes increased latency and reduced bandwidth for important communication operations. Because different applications may be running at any given time, network contention varies and can result in large run-to-run performance variations for certain applications.

Our QoS proposal attempts to address these issues by classifying application traffic into one of several traffic classes based on performance requirements. Each of our class has a unique arbitration priority, an assured bandwidth limit, and a peak bandwidth limit to efficiently regulate access to network channels. We demonstrated how our solution can be used to define four traffic classes – *Low latency*, *Best effort*, *Bulk data*, and *Scavenger* – to support the diverse traffic on HPC systems and improve application performance.

Our proposal ensures consistent, low-latency performance for latency-sensitive traffic, achieving near-baseline performance for MPI_Allreduce operations with different jobs scales and node placements. Our solution also provides the ability to maintain high throughput in the best effort class, securing sufficient bandwidth for applications in order to guarantee overall system throughput. Classes can also be tuned to constrain certain traffic from monopolizing the network while still being allowed to progress at predefined rates.

Our solution's flexibility in provisioning multiple QoS classes with explicit, tunable assured and peak rate limits allows individual HPC sites to tailor class settings for their needs. Furthermore, the use of QoS stall metrics can isolate the adversarial traffic-to-class assignments and help tune the configuration, deployment, and management of QoS in production.

References

[1] S. A. Smith, C. E. Cromey, D. K. Lowenthal, J. Domke, N. Jain, J. J. Thiagarajan, and A. Bhatele, "Mitigating Inter-Job Interference Using Adaptive Flow-Aware Routing," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 346–360.

[2] K. A. Brown, N. Jain, S. Matsuoka, M. Schulz, and A. Bhatele, "Interference between I/O and MPI Traffic on Fat-tree Networks," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–10.

[3] S. A. Smith, "UNDERSTANDING AND MITIGATING NETWORK INTERFERENCE ON HIGH-PERFORMANCE COMPUTING SYSTEMS," Ph.D., University of Arizona, Jun. 2020.

[4] D. D. Sensi, S. D. Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An In-Depth Analysis of the Slingshot Interconnect," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*, Nov. 2020.

[5] P. L. Dordal, *An Introduction to Computer Networks*, Aug. 2020.

[6] A. Jokanovic, J. C. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg, "Effective Quality-of-Service Policy for Capacity High-Performance Computing Systems," in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, Jun. 2012, pp. 598–607.

[7] OFI Working Group, *Libfabric Programmer's manual*, 2020 (accessed Oct 19, 2020). [Online]. Available: https://ofiwg.github.io/libfabric/master/man/fi_endpoint.3.html

[8] Hewlett Packard Enterprise, *Shasta Software Workshop*, 2019 (accessed Oct 19, 2020). [Online]. Available: https://cug.org/proceedings/cug2019_proceedings/includes/files/inv113s1-file1.pdf

[9] M. Mubarak, N. McGlohon, M. Musleh, E. Borch, R. B. Ross, R. Huggahalli, S. Chunduri, S. Parker, C. D. Carothers, and K. Kumaran, "Evaluating quality of service traffic classes on the megafly network," in *High Performance Computing*. Cham: Springer International Publishing, 2019, pp. 3–20.

[10] J. Wilke and J. Kenny, "Opportunities and limitations of quality-of-service in message passing applications on adaptively routed dragonfly and fat tree networks," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020.

[11] L. Savoie, D. K. Lowenthal, B. R. de Supinski, K. Mohror, and N. Jain, "Mitigating inter-job interference via process-level quality-of-service," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–5.

[12] R. E. Grant, K. T. Pedretti, and A. Gentile, "Overtime: a tool for analyzing performance variation due to network interference," in *Proceedings of the 3rd Workshop on Exascale MPI*, ser. ExaMPI '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2831129.2831133

[13] T. Groves, Y. Gu, and N. J. Wright, "Understanding Performance Variability on the Aries Dragonfly Network," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 809–813, iSSN: 2168-9253.

[14] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "Gpcnet: Designing a benchmark suite for inducing and measuring contention in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[15] T. I. Society, "A Two Rate Three Color Marker," https://tools.ietf.org/html/rfc2698, 1999, online; accessed 01 June 2020.

[16] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, "Codes: Enabling co-design of multilayer exascale storage architectures," 2011.

[17] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: a high-performance, low memory, modular time warp system," in *Proceedings Fourteenth Workshop on Parallel and Distributed Simulation*, 2000, pp. 53–60.

[18] Hewlett Packard Enterprise, "Measuring Network Performance to Better Manage IT," Technical White Paper a50002193ENW, Aug. 2020.

[19] Y. Zhang, O. Tuncer, F. Kaplan, K. Olcoz, V. J. Leung, and A. K. Coskun, "Level-spread: A new job allocation policy for dragonfly networks," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 1123–1132.

[20] John Thompson, *Scalable Workload Models for System Simulations*, 2014 (accessed Oct 19, 2020). [Online]. Available: https://hpc.pnl.gov//modsim/2014/Presentations/Thompson.pdf

[21] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of mpi usage on a production supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[22] S. Wienke, J. Miller, M. Schulz, and M. S. Müller, "Development effort estimation in hpc," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 107–118.

[23] L. Savoie, D. K. Lowenthal, B. R. de Supinski, and K. Mohror, "A Study of Network Quality of Service in Many-Core MPI Applications," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 1313–1322.