# Evaluating Quality of Service Traffic Classes on the Megafly Network

Misbah Mubarak[1(✉)], Neil McGlohon[3], Malek Musleh[2], Eric Borch[2], Robert B. Ross[1], Ram Huggahalli[2], Sudheer Chunduri[4], Scott Parker[4], Christopher D. Carothers[3], and Kalyan Kumaran[4]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
mmubarak@anl.gov
[2] Intel Corporation, Santa Clara, CA, USA
[3] Rensselaer Polytechnic Institute, Troy, NY, USA
[4] Argonne Leadership Computing Facility (ALCF), Argonne National Laboratory, Lemont, IL, USA

**Abstract.** An emerging trend in High Performance Computing (HPC) systems that use hierarchical topologies (such as dragonfly) is that the applications are increasingly exhibiting high run-to-run performance variability. This poses a significant challenge for application developers, job schedulers, and system maintainers. One approach to address the performance variability is to use newly proposed network topologies such as megafly (or dragonfly+) that offer increased path diversity compared to a traditional fully connected dragonfly. Yet another approach is to use quality of service (QoS) traffic classes that ensure bandwidth guarantees. In this work, we select HPC application workloads that have exhibited performance variability on current 2-D dragonfly systems. We evaluate the baseline performance expectations of these workloads on megafly and 1-D dragonfly network models with comparably similar network configurations. Our results show that the megafly network, despite using fewer virtual channels (VCs) for deadlock avoidance than a dragonfly, performs as well as a fully connected 1-D dragonfly network. We then exploit the fact that megafly networks require fewer VCs to incorporate QoS traffic classes. We use bandwidth capping and traffic differentiation techniques to introduce multiple traffic classes in megafly networks. In some cases, our results show that QoS can completely mitigate application performance variability while causing minimal slowdown to the background network traffic.

## 1 Introduction

With modern high-performance computing (HPC) systems shifting to hierarchical and low-diameter networks, dragonfly networks have become a popular choice. They have been deployed in multiple high-performance systems including Cori, Trinity, and Theta systems at NERSC, Los Alamos National Laboratory,

M. Weiland et al. (Eds.): ISC High Performance 2019, LNCS 11501, pp. 3–20, 2019.
https://doi.org/10.1007/978-3-030-20656-7_1

and Argonne National Laboratory, respectively [2,11,16]. Dragonfly is a hierarchical topology that uses short electrical links to form groups of routers using a 1-D or 2-D all-to-all interconnect. These groups are then connected all-to-all via optical links. While this design offers low diameter and cost, it increases contention for the link bandwidth among multiple applications which introduces performance variability [5]. For next-generation systems, HPC designers are considering variations of the dragonfly topology that offer increased path diversity, fairness, and scalability [17]. One such topology that has been recently proposed is the dragonfly+, or megafly, which uses a two-level fat tree to form groups of routers. These groups are then connected all-to-all via optical links. Megafly networks use the path diversity of a two-level fat tree to alleviate the communication bottlenecks that can be introduced with standard dragonfly networks. Megafly networks also have the added advantage of using only two virtual channels (VCs) for deadlock prevention as opposed to up to four virtual channels used in a fully connected 1-D dragonfly network. Prior work [8] has shown that while the design of megafly networks helps mitigate performance variability to some extent, it does not completely eliminate it.

Although quality of service (QoS) has been investigated and implemented on TCP/IP networks and data-centers [3], the mechanism remains largely unexplored in the context of HPC networks. Our work is one of the early studies to investigate the role of QoS traffic classes in reducing performance variability caused by communication interference on the now popular hierarchical networks. In this work, we use HPC application workloads that demonstrate performance variability on current dragonfly systems as shown by Chunduri et al. [5]. We replay the workloads on CODES packet-level interconnect simulations [14,20] to answer questions about dragonfly and megafly network topologies: How does the performance of a megafly network compare with a fully connected dragonfly network? How do traffic classes help with performance variability on a megafly network?

The contributions of this work are as follows. (1) We evaluate the performance variability of HPC application workloads on both a megafly and a 1-D dragonfly network using similar network configurations. We compare the performance of a megafly network with a 1-D dragonfly to determine whether megafly network is better resistant to perturbation. (2) We exploit the fact that megafly requires fewer virtual channels for deadlock prevention (as compared to conventional dragonfly), and we use the unused VCs to introduce QoS traffic classes. We evaluate two mechanisms through which QoS can be introduced in HPC networks. First, using bandwidth capping and traffic prioritization, we quantify the impact of QoS when an entire high-priority traffic class is dedicated to an application or set of applications. Second, we dedicate the high-priority traffic class to latency-sensitive operations such as MPI collectives and observe the performance improvement. (3) We extend the CODES simulation framework to perform packet-level simulation of HPC networks in an online mode driven by the scalable workload models (SWM) [18] for use in the above-mentioned experiments.

## 2 Exploring Quality of Service on HPC Networks

In the past, HPC systems were often constructed with torus networks, and jobs were allocated onto partitions of the network that reduced resource sharing and communication interference. With hierarchical networks sharing resources such as switches and links, partitioning becomes more difficult and introducing traffic classes becomes an important step to mitigate communication interference. The slowdown caused by communication interference can significantly impact the overall application performance as the typical range of communication time in communication intensive applications is in the range of *50–80%* [6]. Significant performance variability due to communication interference has been reported in [5], where a slowdown of up to 2x is seen on a production system.

Current 2-D dragonfly networks use up to 4 virtual channels to prevent deadlocks, which is typically all the VCs available. Megafly networks use only 2 VCs for deadlock prevention, thus making them a better candidate for enabling multiple traffic classes. Figure 1 shows one way to implement quality of service on HPC networks. In this implementation, a bandwidth monitor component in each switch tracks the bandwidth consumption of each traffic class for every port. The bandwidth monitoring is done over a static time window $t_w$. Each traffic class is assigned a certain fraction of maximum available link bandwidth, which serves as the upper bandwidth cap for that traffic class while the link is oversubscribed. If the bandwidth consumption of a traffic class reaches the cap and the link is oversubscribed, the traffic class is designated as inactive for the remaining duration of the static window $t_w$. An inactive traffic class has the lowest priority and it gets scheduled only if there are no packets in the remaining higher priority traffic classes. At the start of the window $t_w$, the bandwidth statistics for each traffic class are reset to zero, and the traffic class(es) marked as inactive are activated again. If all the traffic classes are violating their bandwidth cap, then a round-robin scheduling policy is used for arbitration.

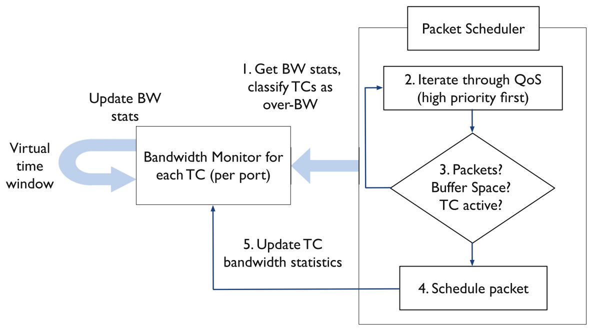

Packet Scheduler
1. Get BW stats, classify TCs as over-BW
Update BW stats
2. Iterate through QoS (high priority first)
Virtual time window
Bandwidth Monitor for each TC (per port)
3. Packets? Buffer Space? TC active?
5. Update TC bandwidth statistics
4. Schedule packet

**Fig. 1.** Enabling quality of service on HPC networks (TC – traffic class, BW – bandwidth, QoS – quality of service)

The implementation of QoS can be beneficial in reducing communication interference on hierarchical networks. For instance, a common problem exhibited on such networks is that communication-intensive (or bandwidth-hungry) applications can "bully" less-communication-intensive applications [20]. With QoS-enabled networks, bandwidth-hungry applications can be prevented from exceeding their permissible bandwidth limits. This approach allows less-communication-intensive applications to have their fair share. Alternatively, one can assign a high-priority traffic class to latency-sensitive operations such as MPI collectives. We report on experiments with both of these QoS mechanisms in Sect. 5.

## 3 Evaluation Methodology

In this section, we discuss the simulation environment, network configurations, workloads, rank-to-node mapping policies, and routing algorithms used in the study.

### 3.1 HPC Simulation Environment

The CODES simulation framework provides high-fidelity, massively parallel simulations of prototypical next-generation HPC architectures. The framework has been extensively used for performance analysis of modern interconnect topologies (fat tree, torus, dragonfly, express mesh and slim fly) [14]. The network models have been validated against real architectures [15]. Prior to this work, the CODES simulation framework supported system simulations with post mortem communication traces. Although traces can illustrate realistic system behavior (for a given problem size), their use inhibits flexibility and simulation scalability as compared to other workload representations. Therefore, we extended the CODES simulation suite to replay workloads in an online or in situ mode using the Scalable Workload Models (SWMs) presented in [18].

Scalable workload models are a workload representation approach that focuses on representing the communication patterns, dependencies, computation-communication overlap, and algorithms. The SWM code[1] is decoupled from the original application code as well as from any particular simulator, enabling use across different simulation environments. The SWM runtime supports a set of low-level API communication primitives to support a number of MPI-based communication operations. The primitives used by the SWM closely resemble those of MPI and SHMEM, but they are not constrained to specific syntax or semantics. In this paper, we utilize several SWM representations for multiple HPC codes including Nekbone, LAMMPS and nearest neighbor [18].

[1] The Scalable Workload Models code is available at the git repo: https://xgitlab.cels.anl.gov/codes/workloads.git.

## 3.2 Topology and Routing Description

The dragonfly network topology, proposed by Kim et al. [10], consists of groups of routers that are connected to each other with one or more optical channels. Within each group, the routers are directly connected to each other in an all-to-all manner via electrical links. In this paper, we refer to this configuration as a 1-D dragonfly. A variation of a dragonfly topology, deployed in the Cray XC systems, uses a 2-D all-to-all within each group instead of all-to-all connections. We refer to this configuration as a 2-D dragonfly. A 2-D dragonfly traverses almost double the number of hops as a 1-D dragonfly. The hop count traversed in a 1-D dragonfly is close to that of a megafly network. Therefore, we compare megafly with a 1-D dragonfly to ensure a reasonable comparison. Various forms of adaptive routing have been proposed for a dragonfly, which detect congestion and determine whether the packet should take a minimal or non-minimal route. We use the progressive adaptive routing algorithm (PAR) provided in [19]. The PAR algorithm in the simulation re-evaluates the minimal path until either the packet decides to take a nonminimal route or the packet reaches the destination group on a minimal path. In this work, we use four virtual channels for progressive adaptive routing in a dragonfly network, as suggested in [19].

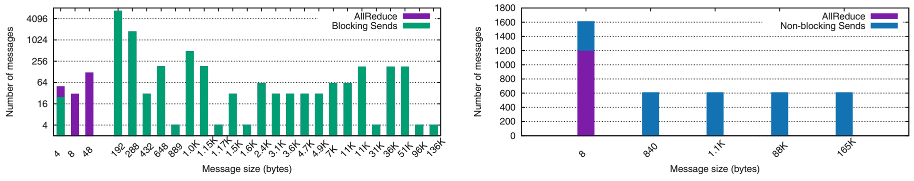
What separates various dragonfly topologies from each other is largely based on the interconnect within a group. Megafly is a topology that belongs to the Dragonfly class of interconnection networks. At a high level, it is classified as having groups of routers which are, in turn, connected to each other with at least one global connection between any two groups. Megafly is characterized by its connectivity in the form of a two-level Fat Tree network in each group. This locally defined network is also known as a complete bipartite graph: a graph with two sub-groups where all nodes within one subgroup are connected to all nodes in the other subgroup. There are no connections between the routers within the same subgroup. The two levels in each group have routers that will be referred to as *Leaf Routers*, those that have terminal/compute node connections but no global connections, and *Spine Routers*, those that have global connections to other groups but no terminal/compute node connections [8,17]. In this paper, we use the progressive adaptive routing algorithm proposed in prior studies on Megafly networks [8]. Megafly requires only two virtual channels (VCs) to avoid deadlock and none to avoid congestion in the intermediate group.

## 3.3 Network Configurations

To perform a comparison of the megafly network with a 1-D dragonfly, we maintain similar router radix and similar node counts. We used a router radix of 32 ports for both networks. Across the group, the routers are connected via global channels. The configurations of the 1-D dragonfly and megafly are given in Table 1.

**Table 1.** Configurations of megafly and 1-D dragonfly used for performance comparison. Link bandwidth for each network is 25 GiB/s (GC - Global Channels)

| | Radix | Groups | Nodes/Group | Node count | GC/Group | Nodes per router |
|---|---|---|---|---|---|---|
| Megafly | 32 | 33 | 256 | 8448 | 256 | 16 (Leaves Only) |
| 1D Dragonfly | 32 | 65 | 128 | 8320 | 128 | 8 |



(a) LAMMPS (b) Nekbone

**Fig. 2.** Message distributions for Nekbone and LAMMPS application workloads

## 3.4 Workloads

In order to quantify the slowdown of a particular job due to communication interference, multiple jobs need to be running in parallel to exhibit interference. We conduct two types of interference experiments: (i) replay HPC applications that serve as foreground communication traffic in conjunction with a job that generates synthetic background communication to understand the interference in a controlled manner, and (ii) replay multiple HPC applications in parallel to capture the dynamism of multi-phased communication and quantify the impact of perturbation.

**Foreground Traffic.** Previous work demonstrates the performance variability shown by LAMMPS, Nekbone, and MILC applications on the Cray XC40 system [5]. Thus, we use LAMMPS and Nekbone workloads as foreground workloads for our experimental analysis. We also use a 3-D nearest-neighbor communication pattern, which is a commonly used pattern in several HPC applications.

LAMMPS is a large-scale atomic and molecular dynamics code that uses MPI for communication. We use the SWM code that derives its communication pattern from the LAMMPS application. Figure 2(a) shows the message distribution of LAMMPS SWM per rank in a problem involving 2,048 ranks. The LAMMPS workload uses MPI_AllReduce with small messages as well as blocking sends and nonblocking receives for point-to-point communication with large messages.

Nekbone is a thermal hydraulics mini-app that captures the structure of the computational fluids dynamics code Nek5000. Nekbone's SWM communication pattern is derived from the Nekbone benchmark. Figure 2(b) shows the message distribution of the Nekbone SWM on a per rank basis in a problem with 2,197 ranks. Nekbone performs a large number of MPI collective operations with small 8-byte messages. It uses nonblocking sends and receives to transmit medium-sized messages.

Cartesian neighborhood communication is a pattern commonly used in multiple scientific applications including Hardware Accelerated Cosmology Code (HACC), fast Fourier transform solvers, and adaptive mesh refinement (AMR) codes. We use a 3-D nearest-neighbor SWM in this work that transmits large messages (64 KiB and 128 KiB) on a per rank basis with multiple iterations of MPI nonblocking sends and receives followed by MPI_Wait_All. A problem size of 4,096 ranks is used with the nearest neighbor SWM.

**Background Traffic.** The background communication traffic is needed to interfere with the foreground workloads. To ensure an even distribution of traffic that covers a significant fraction of the network, we use a uniform random communication pattern. This is generally considered a benign traffic pattern for dragonfly networks. However, with large messages randomly sent in the network, uniform random causes hotspots at multiple network locations and becomes a source of interference. We varied the amount of data transmitted via uniform random traffic and observed the effect of different data transmission rates on the foreground traffic. The background traffic generation is modeled as a separate job that runs in parallel with the foreground traffic and occupies at least 25% to 50% of the entire network. The background injection rates depend on the available compute node to router link bandwidth in the network. We inject traffic at a percentage of the available link bandwidth and vary the injection rates between 2% to 36.5% of the link bandwidth. At the 36.5% rate, each node is injecting 9 GiB/s of background traffic with an aggregate network background interference of 18 TiB/s. At this rate, we see significant slowdown (up to *4x* for uniform random and up to *7x* for random permutation) in application communication times for both networks. Therefore, we keep that as the maximum background injection rate.

We experiment with two different background communication patterns: (i) a uniform random synthetic pattern where a rank randomly chooses a destination rank and transmits large messages and (ii) a random permutation traffic where a pair of ranks communicate and transmit data until a certain threshold is reached.

**Multiple Applications.** As a specific instance of representative HPC scenarios, we ran the three foreground workloads in parallel (Nekbone, nearest neighbor, and LAMMPS). We also ran each of these workloads in isolation on the network to determine the baseline performance and observed the slowdown introduced when the workloads are running in parallel.

### 3.5 Rank-to-Node Mappings

Ranks are placed on network nodes in a manner similar to that for production HPC systems, where clusters of available network nodes are assigned to a job. Therefore, we use a geometric job placement policy in which multiple clusters of network nodes are assigned to jobs. In the simulation, the clusters are formed by using the inverse transform sampling method for creating random samples from a given distribution. The experiments in the paper were performed with three different rank to node mappings; however, we did not observe a noticeable difference between the statistics reported by each mapping.

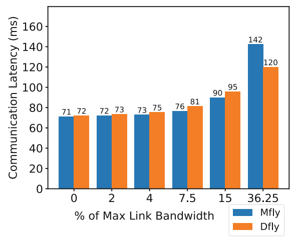
## 4 Quantifying Interference on 1-D Dragonfly and Megafly Networks

In this section, we analyze the communication interference on both 1-D dragonfly and megafly networks following the methodology described in Sect. 3. Each simulation experiment was conducted three times with different geometric job allocation policies and the performance difference observed between each run was less than 2%. The communication latency of an application is determined by the rank that incurs the maximum latency across all the participating ranks. Given that the distribution of the latencies of ranks potentially contain long tails [13] corresponding to ranks that are effected by congestion, using maximum latency across the ranks is the appropriate metric. Before discussing the QoS experiments, we compare the baseline communication performance of megafly and 1-D dragonfly networks and further quantify the performance degradation with QoS disabled. We then incorporate QoS mechanisms in the megafly networks to evaluate the performance with and without such mechanisms enabled.

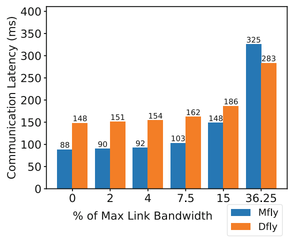
**Uniform Random Background Traffic.** Figure 3(a) shows the communication time of LAMMPS SWM with varying degrees of background traffic, starting from no background traffic, on both megafly and 1-D dragonfly. LAMMPS uses a mix of point-to-point and collective communication as shown in Fig. 2. The performance results in the figure show that megafly performs better than a 1-D dragonfly in most of the background traffic injection rates. For the worst case background injection rate, 1-D dragonfly outperforms megafly.

Figure 3(b) shows the communication time of Nekbone SWM with and without uniform random background traffic on both megafly and 1-D dragonfly networks. Nekbone uses a large number of 8-byte MPI collectives as shown in Fig. 2. Additionally, of the studied workloads, Nekbone is the most communication volume intensive application: it transmits 4x more data than the LAMMPS or nearest-neighbor SWM workloads do. The performance results in the figure show that megafly performs up to 60% better than a 1-D dragonfly in all except one background traffic injection rates. For the worst case background injection rate, 1-D dragonfly outperforms megafly.

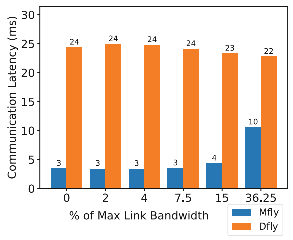
Figure 3(c) shows the performance of nearest-neighbor communication on both 1-D dragonfly and megafly networks. Since we are using geometric job mapping that allocates cluster of network nodes, the nearest-neighbor pattern involves extensive communication between two groups. In this case, megafly consistently outperforms the dragonfly network because of multiple reasons: (i) megafly has larger group sizes (more nodes available within a group), which increases locality of communication within a group. The locality of communication is beneficial for nearest neighbor traffic, and (ii) Dragonfly has a single



(a) LAMMPS communication times comparison



(b) Nekbone communication times comparison

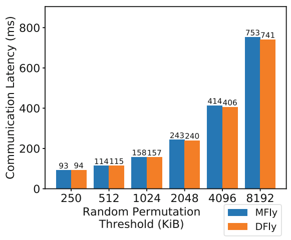


(c) Nearest-neighbor communication times comparison

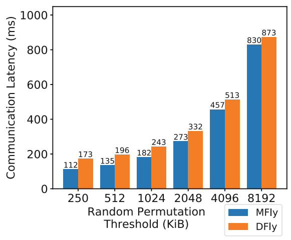
**Fig. 3.** Performance of megafly vs. 1-D dragonfly with (a) geometrically allocated 2048/2048 ranks for LAMMPS/uniform random workloads and (b) geometrically allocated 2197/2197 ranks for Nekbone/uniform random workloads, and (c) geometrically allocated 4096/4096 ranks for nearest-neighbor/uniform random workloads. The intensity of the background traffic was scaled at a percentage of the maximum link capacity.

minimal path between two routers within a group whereas megafly has 16 different minimal path options for this route, which reduces intra-group congestion. Since nearest-neighbor communication exchanges are based exclusively on point-to-point operations between two groups, it is less impacted by the background traffic.

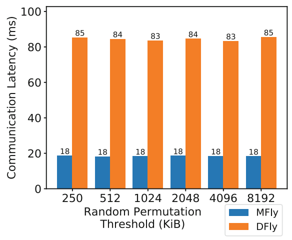
**Random Permutation Background Traffic.** While uniform random traffic with large message sizes can cause dynamic hotspots in the network, we also considered random permutation traffic to introduce more persistent network interference. Similar to uniform random, the random permutation background traffic pattern sends packets to a randomly selected node in the network. We use a rotating random permutation pattern that will send continually to the same randomly selected destination (on a per node basis) until a certain number of bytes have been transmitted before choosing a new random destination. Figures 4 shows the performance of megafly and dragonfly networks. The foreground workloads see a slowdown in communication time as the number of bytes exchanged in the background traffic is increased. Since nearest neighbor traffic involves point to point operations (mostly to the neighboring group), it is not significantly impacted by the rotating random background traffic. While the performance of LAMMPS workload is comparable on both networks, the Nekbone workload is less perturbed on a megafly network than a 1-D dragonfly. Our conjecture is that the better performance of megafly can be attributed to the additional path diversity of its minimal routes. The results demonstrate that there is nearly a linear slowdown in the performance of the foreground job as the number of bytes exchanged in the background traffic increases.



(a) LAMMPS communication times comparison



(b) Nekbone communication times comparison



(c) Nearest-neighbor communication times comparison

**Fig. 4.** Performance of megafly vs. 1-D dragonfly with (a) geometrically allocated 2048/2048 ranks for LAMMPS/random permutation workloads (b) geometrically allocated 2197/2197 ranks for Nekbone/random permutation and (c) geometrically allocated 2197/2197 ranks for nearest-neighbor/Random Permutation workloads. The amount of data exchanged between two nodes in a rotating random permutation was scaled from 250 KiB to 8 MiB.

**Multiple Applications in Parallel.** In the third case, as a specific instance of representative HPC scenarios, we run the three workloads (LAMMPS, Nekbone, and nearest neighbor) in parallel without any synthetic communication traffic. This scenario clearly mimics a common system state, with multiple jobs completing for shared resources. Figure 5 shows the communication time of the three applications when running in parallel and in isolation on both dragonfly and megafly networks. We can see that with both LAMMPS and Nekbone, the applications are much less perturbed on a megafly network than on a 1-D dragonfly network.

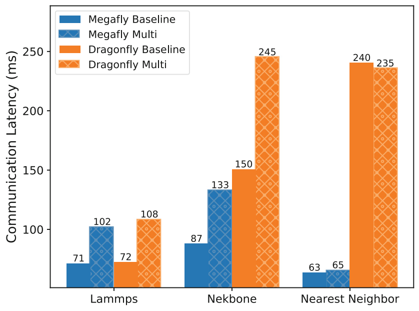


**Fig. 5.** Communication times of LAMMPS (2,048 ranks), Nekbone (2,197 ranks), and nearest neighbor (2,048 ranks) when running in parallel on 1-D dragonfly and dragonfly+ networks. Baseline indicates the application runs in isolation.

*After introducing different forms of background communication traffic with foreground workloads, our analysis shows that in maximum cases, the* *performance of megafly network is comparable to a 1-D dragonfly network. In majority of the cases, Megafly performs better than a 1-D dragonfly. For LAMMPS and Nekbone workloads with worst case background traffic, a 1-D dragonfly gets relatively less perturbed than a megafly. On both networks, however, HPC applications see a significant slowdown in communication ranging up to 700% in the presence of intense background communication traffic.*

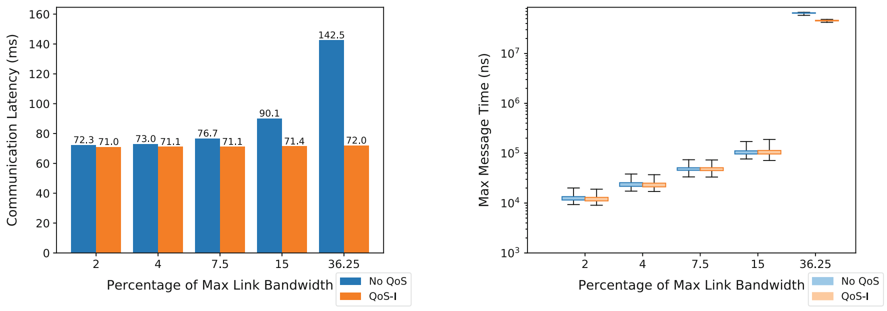
## 5 Evaluating Quality of Service on Megafly Networks

Enabling quality of service on HPC networks requires that each traffic class have its own set of virtual channels. Megafly networks require a fewer number of virtual channels for deadlock prevention. When a fixed, limited number of VCs are available in the switch hardware, megafly needs half as many VCs as a dragonfly and has the opportunity to use the extra VCs for QoS. The mechanism for quality of service was introduced in Sect. 2. In this section, we perform experiments to analyze the impact of QoS on traffic interference and application slowdown that was seen in Sect. 4. Due to space constraints, the results shown are for Megafly networks even though a 1-D dragonfly network performs in a similar manner with QoS turned on. Since there can be a large number of permutations for bandwidth caps, we performed a sensitivity analysis by sweeping different bandwidth values and picked the values that were most effective. The static window over which the bandwidth statistics were monitored was kept to 5 ms throughout these experiments. We explore two configurations through which QoS can be introduced in megafly networks:

### 5.1 QoS Mechanism I: Prioritizing Entire Applications

With our first QoS mechanism, a higher priority and high bandwidth are assigned to the entire application (or set of applications) so that they face minimal slowdown relative to other traffic. We use uniform random background traffic and both LAMMPS and Nekbone foreground workloads that exhibited slowdown on megafly networks in Sect. 4 (Nearest neighbor was not getting significantly perturbed). To understand the impact on background traffic, we measure the performance of both foreground workload and background traffic. The background traffic performance is measured by the maximum time to complete a message (all messages have the same size in the synthetic workload).

The benefit of using this QoS approach is that if the foreground application is not utilizing the full bandwidth allocated to it, then the background workload can consume the unutilized bandwidth. Figure 6 compares the performance of LAMMPS workload with and without QoS enabled on a megafly network. It also shows the slowdown to background communication traffic. The LAMMPS workload is not as communication intensive because it involves point-to-point messages along with a small number of MPI_AllReduce messages. Therefore, the perturbation to background traffic is not significant. Because of the high priority given to LAMMPS, it does not see any slowdown even though it is running in parallel with intense background traffic. Additionally, while we observe a significant speedup with LAMMPS, the background traffic observes only a small degree of slowdown as compared with the no-QoS case.



(a) LAMMPS communication times (b) Background traffic performance

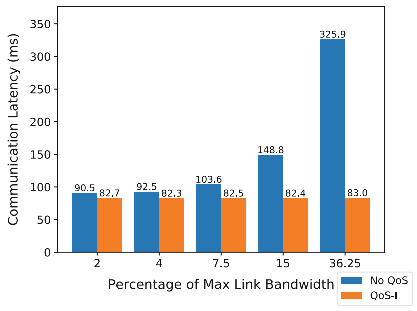
**Fig. 6.** QoS Mechanism I (Application Priority): performance of LAMMPS and background traffic on megafly network with QoS enabled and disabled. The entire LAMMPS application is given a high priority and high bandwidth (70%).

Nekbone SWM is a communication-intensive workload that transmits 4x more data than does LAMMPS SWM, and a majority of the communication involves collectives. Figure 7 shows the performance of the Nekbone SWM when QoS is enabled. Once again we see Nekbone having minimal slowdown when QoS is enabled while causing minimal slowdown to background communication traffic. The primary reason for the improved performance is that both Nekbone and LAMMPS are given high priority and high bandwidth yet they do not consume all the bandwidth assigned to them. Therefore, the background traffic is able to get the required bandwidth that it needs while observing little slowdown.

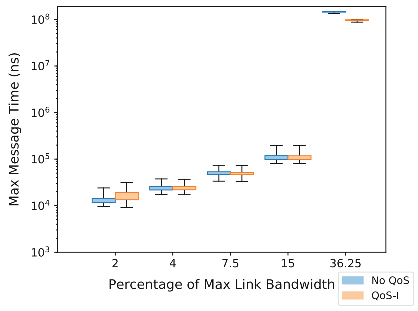
*Both these results demonstrate that traffic differentiation with bandwidth shaping and prioritization can mitigate (or eliminate) communication interference to HPC workloads while causing minimal slowdown to the background traffic. Assigning a high priority to an application can eliminate the perturbation to that application while experiencing a reasonable slowdown in the remaining network traffic.*

### 5.2 QoS Mechanism II: Prioritizing and Guaranteeing Bandwidth to Latency-Sensitive Operations

Several HPC applications rely on the performance of MPI collective operations. In a majority of the cases, collectives comprise small messages, and the application performance suffers when heavy background network traffic interferes with the transmission of these messages. An alternative application of QoS is to assign a high priority and guaranteed bandwidth to collective operations.

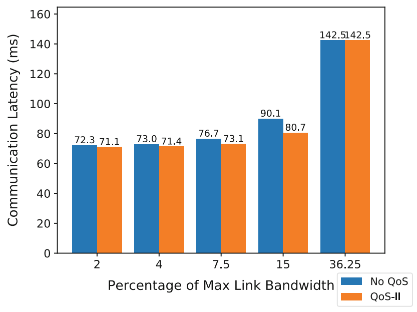


(a) Nekbone communication times

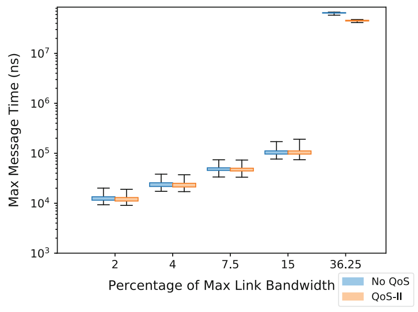


(b) Background traffic performance

**Fig. 7.** QoS Mechanism I (Application Priority): performance of Nekbone and background traffic on megafly network with QoS enabled and disabled. The entire Nekbone application is given a high priority and high bandwidth (70%).



(a) LAMMPS communication times

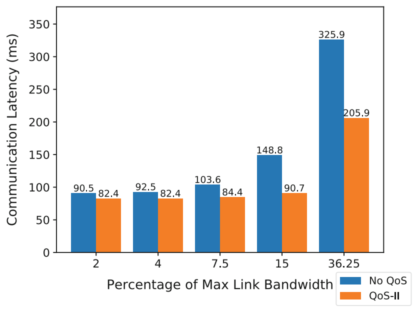


(b) Background traffic performance

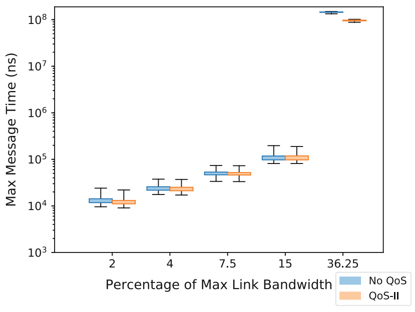
**Fig. 8.** QoS Mechanism II (Collective Priority): performance of LAMMPS and background traffic on megafly network with application-based QoS enabled and 10% bandwidth guaranteed to collectives.

Figure 8 shows the performance of LAMMPS when high priority is given to collectives and compares it with the case where no QoS is enabled. In this case, we are assigning a high priority but a small fraction of bandwidth to collective operations; the point-to-point operations and background traffic are given a lower priority and higher bandwidth cap (90%). We see that although there is some slowdown in foreground traffic when the background traffic becomes intense, the foreground workload is still 10% faster than the case where no QoS is enabled (specifically in the case of 15% background traffic injection). LAMMPS uses more point-to-point operations and has fewer collective operations, which is why the speedup is not as significant as Nekbone that relies heavily on collective performance.

Figure 9 shows the performance of Nekbone when given high priority and a guaranteed bandwidth to collectives. Nekbone relies heavily on collective



(a) Nekbone communication times



(b) Background traffic performance

**Fig. 9.** QoS Mechanism II (Collective Priority): performance of Nekbone and background traffic on megafly network with application-based QoS enabled and 10% bandwidth guaranteed to collectives.

operations. *Therefore, we see a significant performance improvement of up to 60% speedup in communication time compared with the case where no QoS is enabled.* The background communication traffic does not show a slowdown in message communication times; instead it shows a slight performance improvement compared to no-QoS options in one case.

## 5.3 Applying QoS Mechanisms to Multiple Application Workloads in Parallel

In this section, we examine both QoS mechanisms in the case where multiple applications are running in parallel, which is a specific instance of a representative HPC system. We compare the QoS-enabled performance with the case where there are multiple applications running without any QoS. For the first QoS mechanism, since Nekbone is more communication intensive than LAMMPS and nearest neighbor (shown in Fig. 2) we assign it a separate traffic class with a bandwidth cap of 30% and a high priority. The rest of the bandwidth is available to both LAMMPS and nearest neighbor. For the second QoS mechanism, we assign a higher priority to all collective communication in both LAMMPS and Nekbone and then see the impact on application performance.

Figure 10 shows the performance of multiple applications running in parallel with and without QoS enabled. In short, both schemes are beneficial, and lead to reduced communication time. With the QoS Mechanism I, we give a high priority and assign one third of link bandwidth as a cap to Nekbone. Nekbone is communication intensive; and with a high priority and the bandwidth cap, it does not get any slowdown due to background communication traffic. In contrast, LAMMPS and nearest neighbor have a lower priority, and they still see a performance improvement compared with the case where there was no QoS enabled. *Adding bandwidth caps on Nekbone (which is a bandwidth-intensive application) helps improve the performance of LAMMPS and nearest neighbor as well.*

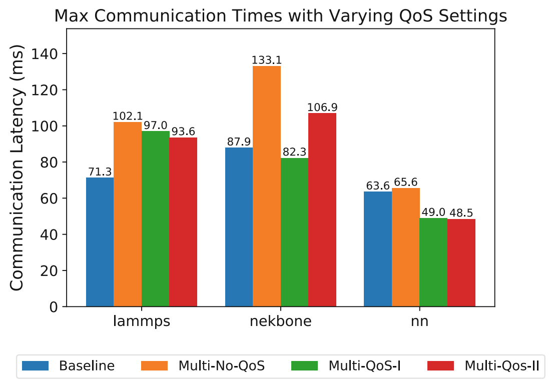


**Fig. 10.** Communication times of Nekbone, LAMMPS, and nearest-neighbor workloads when running in parallel. Both mechanisms of application-based QoS were enabled. The comparison is done with (i) the worst case when no QoS is enabled (Multi No QoS) and (ii) the best case when the workload is running in isolation with no interference (baseline).

With the QoS Mechanism II, where collective communication is given priority, both LAMMPS and Nekbone benefit by seeing a 10% and 20% speedup, respectively, compared with the case where QoS is not enabled. One interesting observation is the performance of the nearest-neighbor workload, which is much faster with QoS enabled than when the workload is running in isolation. Looking at the adaptive routing statistics, we see that the nearest-neighbor workload when running in isolation takes the maximum number of minimal routes because of the bias toward minimal routes. With QoS enabled, nearest-neighbor traffic has a lower priority with QoS mechanisms enabled, which causes it to take more nonminimal routes, coincidentally helping with the congestion points. Thus the workload sees improved performance. A similar phenomena is observed with Nekbone when it is running with QoS Mechanism I.

*These experiments demonstrate the effectiveness of applying QoS to reduce or eliminate communication interference. With both mechanisms, Nekbone, being more bandwidth intensive, sees a 20% to 350% speedup in communication time compared with the case where QoS is not enabled. LAMMPS sees a 10% to 200% improvement in communication time compared with the case where QoS is not enabled. Nearest neighbor is indirectly impacted by bandwidth capping and sees a performance improvement of 25%. Additionally, all applications (both foreground and background traffic) benefit from QoS; the low priority applications are guaranteed to get a share in bandwidth which avoids performance degradation. The takeaway is that adding traffic differentiation in HPC networks can bring performance improvement to all traffic classes.*

## 6 Related Work

There are different approaches to address run-to-run variability on HPC systems. One approach is based on partitioning the networks and providing an isolated partition for a job. While this approach has successfully worked for low-radix networks such as torus [21], it is a challenge to implement partitioning on networks such as dragonfly or megafly, due to their hierarchical nature. The other approach is QoS, which can be enforced through various mechanisms on data centers and HPC networks. Flow control [7,12] is a high-level approach for avoiding interference in large-scale and datacenter-scale networks which takes a coarser-grained look at data within the network. Alizedah et al. [1] studied the impacts of sacrificing a portion of the total bandwidth while lowering the threshold for congestion sensing to provide a buffer zone within links in an attempt to reduce the overall latency of applications in a datacenter environment. On the algorithmic routing side of QoS implementation, many different approaches exist, from centralized global information methods to distributed routing algorithms with limited or incomplete network information and hierarchical algorithms that bridge the gap between globally and locally available information when making routing decisions. Chen and Nahrstedt [3] presented an overview of various routing algorithms solving different QoS problems for both unicast and multicast applications. Most of the literature available on quality of service is intended for data-centric and TCP/IP networks and does not explore HPC workloads, routing, and flow control mechanisms. Cheng et al. [4] provided high-level details about implementing quality of service on data-centric and HPC networks. Jakanovic et al. [9] provided an efficient QoS policy for HPC systems with InifiBand network (fat tree topology).

## 7 Discussion and Conclusion

With HPC applications showing performance variation on recent hierarchical interconnects, we analyze communication interference for both megafly and dragonfly networks. We extend the CODES parallel simulation framework to replay the communication pattern of HPC applications using the Scalable Workload Models (SWM). We introduce moderate to intense background communication traffic during the execution of these communication workloads and compare the slowdown on megafly network with a 1-D dragonfly network. We demonstrate that performance variability is experienced in both topologies, while observing that in several experiments the performance implication is less severe for megafly.

To further mitigate the variability, we introduce traffic differentiation and quality of service mechanisms and show the results on a megafly network. We explore two different QoS mechanisms for HPC workloads (i) prioritizing and bandwidth capping entire HPC applications (ii) prioritizing and guaranteeing bandwidth to latency sensitive collective operations with small messages. With the first mechanism, performance results show that when a high priority and a bandwidth cap is given to entire HPC applications, it can eliminate performance variability while the rest of the background traffic also sees minimal impact. For the second mechanism, we show that when a small fraction of bandwidth is guaranteed to latency sensitive operations like the MPI collectives, it can mitigate the performance variability by 10% to 60% depending upon the intensity of collective communication in the application. This dramatic performance improvement from QoS on megafly can make up for its shortcomings in high-interference runs with no additional hardware cost. In both cases, we saw that both high and low priority applications have a better performance with QoS than the case with no QoS, which implies that having traffic differentiation is beneficial for all applications on a HPC network as it allows a fair share of bandwidth to each traffic class.

While this work is aimed to provide a proof of concept that QoS is effective in mitigating communication interference for realistic HPC workloads, there are a number of avenues that need to be further explored. First, real HPC systems have tens to hundreds of jobs running. Giving a high priority to more than one HPC application (as shown in QoS mechanism I) can introduce interference within the traffic class, which can slowdown high priority applications. Secondly, one would need to explore how to expose the traffic classes to the MPI interfaces and the job scheduler.

**Acknowledgment.** This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. The work has used resources from the Argonne's Leadership Computing Facility (ALCF), Rensselaer's CCI supercomputing center and Argonne's Laboratory Computing Resource Center (LCRC).

## References

1. Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., Yasuda, M.: Less is more: trading a little bandwidth for ultra-low latency in the data center. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pp. 19–19. USENIX Association (2012)
2. Argonne Leadership Computing Facility (ALCF): Theta, Argonne's Cray XC System. https://www.alcf.anl.gov/theta
3. Chen, S., Nahrstedt, K.: An overview of quality of service routing for next-generation high-speed networks: problems and solutions. IEEE Netw. **12**(6), 64–79 (1998)
4. Cheng, A.S., Lovett, T.D., Parker, M.A.: Traffic class arbitration based on priority and bandwidth allocation. Google Patents, December 2016
5. Chunduri, S., et al.: Run-to-run variability on Xeon Phi based Cray XC systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 52. ACM (2017)
6. Chunduri, S., Parker, S., Balaji, P., Harms, K., Kumaran, K.: Characterization of MPI usage on a production supercomputer. In: Characterization of MPI Usage on a Production Supercomputer. IEEE (2018)

7. Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S.: DevoFlow: scaling flow management for high-performance networks. In: ACM SIGCOMM Computer Communication Review, vol. 41, pp. 254–265. ACM (2011)
8. Flajslik, M., Borch, E., Parker, M.A.: Megafly: a topology for exascale systems. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) ISC High Performance 2018. LNCS, vol. 10876, pp. 289–310. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92040-5_15
9. Jokanovic, A., Sancho, J.C., Labarta, J., Rodriguez, G., Minkenberg, C.: 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 (HPCC-ICESS), pp. 598–607. IEEE (2012)
10. Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-driven, highly-scalable dragonfly topology. In: 35th International Symposium on Computer Architecture 2008. ISCA 2008, pp. 77–88. IEEE (2008)
11. Los Alamos National Laboratory: Trinity Cray XC40 system. http://www.lanl.gov/projects/trinity/
12. McKeown, N., et al.: OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
13. Mubarak, M., et al.: Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 204–215. IEEE (2017)
14. Mubarak, M., Carothers, C.D., Ross, R.B., Carns, P.H.: Enabling parallel simulation of large-scale HPC network systems. IEEE Trans. Parallel Distrib. Syst. **28**(1), 87–100 (2017)
15. Mubarak, M., Ross, R.B.: Validation study of CODES dragonfly network model with Theta Cray XC system (2017). https://doi.org/10.2172/1356812
16. NERSC: Cori. https://www.nersc.gov/users/computational-systems/cori/
17. Shpiner, A., Haramaty, Z., Eliad, S., Zdornov, V., Gafni, B., Zahavi, E.: Dragonfly+: low cost topology for scaling datacenters. In: 2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), pp. 1–8. IEEE (2017)
18. Thompson, J.: Scalable workload models for system simulations background and motivation. Technical report (2014). http://hpc.pnl.gov/modsim/2014/Presentations/Thompson.pdf
19. Won, J., Kim, G., Kim, J., Jiang, T., Parker, M., Scott, S.: Overcoming far-end congestion in large-scale networks. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 415–427. IEEE (2015)
20. Yang, X., Jenkins, J., Mubarak, M., Ross, R.B., Lan, Z.: Watch out for the bully! job interference study on dragonfly network. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC16, pp. 750–760. IEEE (2016)
21. Zhou, Z., et al.: Improving batch scheduling on Blue Gene/Q by relaxing 5D torus network allocation constraints. In: 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 439–448. IEEE (2015)