LLNL-CONF-745381

A Study of Quality of Service in MPI Applications

L. Savoie, D. Lowenthal, B. de Supinski, K. Mohror

January 29, 2018

ROME 2018
Vancouver, Canada
May 25, 2018 through May 25, 2018

A Study of Network Quality of Service in Many-Core MPI Applications

Lee Savoie
David K. Lowenthal
Dept. of Computer Science, The University of Arizona

Bronis R. de Supinski
Kathryn Mohror
Lawrence Livermore National Laboratory

***Abstract*—Network contention in existing high performance computing (HPC) systems increases job execution time and reduces machine throughput. This problem is expected to become worse in future systems as core counts increase and networks become larger and more complicated. In this paper, we investigate the use of network Quality of Service (QoS) to mitigate the effects of network contention. QoS allocates bandwidth to individual jobs, thus limiting the impact that one job can have on another through network contention. We consider coarse-grained QoS, in which each job runs at a different priority level, by running a number of micro-benchmarks and applications in different QoS configurations on real hardware with QoS capabilities. Our results indicate that while network contention reduces job performance by as much as 70%, coarse-grained QoS is unlikely to improve throughput on HPC systems and may increase job execution times by more than 100%. Based on our analysis, finer-grained QoS is more likely to improve performance and throughput.**

I. Introduction

In high performance computing (HPC), several jobs execute concurrently in high-end machines consisting of up to thousands of nodes. The core counts of these high-end machines are rapidly increasing, which means that each node can produce data at a faster rate. This in turn means that it is likely that more data will be communicated between nodes, either because there are more potential message senders and receivers (in a flat MPI model) or because a "leader" thread sends more data (e.g., in an MPI+OpenMP model).

One negative aspect of the ability of many-core nodes to send large amounts of data is that there is more total data on the network, and increases in node count and network capacity may not keep pace with increases in per-node core count. This increases the chance that network congestion due inter-job interference occurs. While schedulers for HPC machines generally use space sharing—in which each job is assigned a disjoint set of nodes to prevent on-node sharing and performance degradation—the network on most HPC machines is shared between jobs. System software solutions to network contention will therefore become increasingly important.

Network contention occurs when multiple processes send messages that travel over the same network link at the same time. When this happens, messages must share the bandwidth of the affected link, which generally reduces the performance of all messages traveling through the link. This, in turn, can degrade the performance of the jobs that sent the affected messages and reduce the throughput of the HPC machine.

One way to mitigate the problems caused by contention is to use Quality of Service (QoS) capabilities that are already built into InfiniBand, a common networking fabric in HPC. QoS is already used in the Internet and Local Area Networks to prioritize different types of traffic, but it is not widely used in HPC installations. QoS mechanisms can allocate bandwidth among different applications (or classes of applications), thus guaranteeing each application or class a minimum amount of bandwidth. This can protect applications, to some degree, from network interference from other applications, which may improve overall throughput.

In this paper, we investigate the use of QoS to improve throughput on HPC machines by running several micro-benchmarks and MPI applications under different QoS configurations. To the best of our knowledge, this is the first empirical study of using QoS to mitigate inter-job interference on HPC hardware. Our results show that network contention increases job execution time by as much as 70%. However coarse-grained QoS, in which each job runs at a different priority level, is unlikely to significantly improve HPC throughput and may increase job execution times by more than 100% in some cases. This differs from previous results based on simulation [1]. We also analyze the performance of our micro-benchmarks to understand why we were unable to improve throughput. In general, we found that assigning any single application a high priority tends to speed up ranks that do not need to be sped up, which unnecessarily slows down other jobs and decreases overall throughput. Our analysis shows that applying QoS at the rank level rather than the job level is more likely to improve job performance and throughput.

The rest of this paper is organized as follows. In Section II we discuss the problem of network contention and provide background on QoS. In Section III we describe our experimental setup, including the micro-benchmarks and applications we ran and the hardware we used for testing. We discuss our results in Section IV and present related work in Section V. Finally, we summarize our findings in Section VI.

II. Background

Network contention is a long-standing problem in HPC. Contention occurs when multiple network flows—either from the same application or different applications—use the same network hardware at the same time. Contention increases the amount of time routers and switches take to process each packet. It also forces network flows to share bandwidth with other flows. Thus, each flow experiences an increase in latency and a decrease in effective bandwidth. In other words, contention causes the network to slow down message transmission. Slower message transmission, in turn, can result in more idle time and longer execution times for applications.

Contention is particularly difficult to analyze and understand because it is unpredictable. The amount of contention that a job experiences is affected by many parameters, including the communication characteristics of all jobs running on the system, the allocation of nodes to jobs, the times at which each job uses the network, and the network routing algorithm. Because contention is context dependent, the same job can experience widely different contention patterns (and thus execution times) when run at different times on the same system.

Network contention is already a problem on HPC systems, and this problem is expected to get worse as we build larger systems. Nodes in exascale systems will likely be much more powerful than nodes in current systems, with (possibly multiple) accelerators and many cores that can process large amounts of data. This will allow users to perform larger and more detailed computations, but it also will likely result in larger and/or more messages being sent across the network. In addition to this, machines with larger numbers of nodes will likely also have larger and more complicated networks. Messages may have to travel more hops to reach their destination, and they may be required to traverse hops with reduced bandwidth. These problems will increase the amount of data on the network, the amount of time that messages spend on the network, and the probability that multiple messages will use the same link at the same time. This will result in more network contention and thus greater performance degradation.

Space sharing, or spatially dividing resources between jobs, is one way to eliminate contention. Space sharing is already typical for nodes; specific nodes are allocated to specific jobs and jobs do not share nodes. This eliminates inter-job contention for on-node resources such as the CPU and memory. Space sharing is also a viable strategy for some kinds of networks; each job is placed in a disjoint part of the network so that no job shares a network link with other jobs. For example, BlueGene systems use network space sharing [2]. However, because of the need to schedule jobs in disjoint sections of the network, space sharing results in reduced machine utilization. Because high utilization is a goal of many HPC installations, most HPC machines do not space share the network. For this reason we are investigating other methods of addressing network contention.

Many modern networks have built-in Quality of Service (QoS) mechanisms that can be used to mitigate the problems of contention. There are many different types of QoS, but in this paper, we are interested in traffic prioritization. A network that uses QoS can assign different priorities to different flows or classes of flows. The priority of each flow determines how it is handled as it is transferred across the network and limits the amount of bandwidth it can consume in the presence of other network traffic. QoS, in the form of differentiated services [3], has existed in the Internet for two decades and has been used in video streaming [4], [5], cloud and data center networks [6], and wireless networks [7], among other things.

QoS has been available in HPC networks for several years. For example, InfiniBand, a network fabric used in many HPC installations, allows users to divide flows into different service levels (SLs), where the SL defines the priority of its traffic. Each SL is associated with a priority, and SLs are allowed to send data in a weighted round-robin fashion [8]. Thus, if SL A has QoS of 3 and SL B has QoS of 5, SL A will be allowed to send 3 packets, then SL B will be allowed to send 5 packets, and so on. In addition to this, each SL is managed independently, so contention on one SL cannot result in blocked packets on another SL, even if they share the same link. Thus, QoS can be used in InfiniBand to limit the amount of network contention that a job can experience.

QoS support is also built in to common MPI libraries. For example, MVAPICH2, a popular MPI implementation, allows users to set the service level for each job using an environment variable. We use this capability in our QoS system.

Despite its availability in common network fabrics and MPI libraries, QoS has not seen widespread use in HPC. Researchers have considered some ways of using QoS to improve performance of HPC jobs, such as dividing traffic into different SLs but assigning the same priority to all SLs [9], [10]. Researchers have also looked into separating traffic into different SLs, but this has only been tested in simulation [1]. In this work we investigate QoS on actual hardware and divide traffic into SLs with different priorities.

III. Experimental Setup

We evaluated the effectiveness of QoS at improving performance and throughput by running several micro-benchmarks and applications in various configurations with various QoS settings. All of the benchmarks and applications are built on MPI. In the following sections, we first describe the micro-benchmarks and applications, and then describe the tests that we ran.

A. Micro-Benchmarks

We used four micro-benchmarks that implement common HPC communication patterns. Each of the benchmarks can be tuned to send different sizes of messages or to compute for different amounts of time. We call the benchmarks *Flood-Pairs*, *Nearest-Neighbor*, *All-to-all*, and *Random-Pairs*. Each is described below in greater detail.

Flood-Pairs is a modified version of the bisection bandwidth test from the CORAL MPI benchmarks [11] which is designed to measure the bisection bandwidth of an HPC installation. We are not using it to measure bisection bandwidth but rather for its communication characteristics. In this benchmark ranks are grouped into pairs, each pair exchanges messages repeatedly, and then the aggregate bandwidth is calculated. We selected this benchmark because of its simplicity and its heavy network usage. In our tests, the lower half of ranks were paired with the upper half of ranks; for example, in a 140 rank test, rank

0 would be paired with rank 70, rank 1 would be paired with rank 71, and so on. Because ranks are assigned in a block fashion in our setup (the first block of ranks is assigned to the first node, etc.), this ensures that as much communication as possible is off-node.

Our second micro-benchmark, *Nearest-Neighbor*, is a simple 2D nearest neighbor exchange that we wrote. It lays out ranks in a 2D grid, and each node exchanges messages with its 4 neighbors in the grid. This continues for a set number of iterations. There are also periodic global barriers after a set number of iterations. We selected this benchmark because nearest neighbor communication is common in MPI applications.

Like *Nearest-Neighbor*, *All-to-all* is a simple benchmark that we wrote. Each iteration is a single `MPI_Alltoall` call in which every rank exchanges data with every other rank. This repeats for a set number of iterations. This benchmark is designed to represent applications that include periodic global synchronization. While all-to-all communication is not a perfect representation of an MPI application with periodic global synchronization, it is a simple approximation that can help us understand how QoS affects communication intensive applications with frequent synchronization.

Our final benchmark, *Random-Pairs*, is similar to *Flood-Pairs* in that the lower half of the ranks are paired with the upper half of the ranks and then pairs send large amounts of data to each other. However, in *Random-Pairs* the pairs are assigned randomly. This allows us to study more complex, irregular communication patterns.

B. Applications

We ran tests with four applications: *Qbox*, *Crystal Router*, *MILC*, and *pF3D*. Each is either an application used for production science or a proxy application designed to mimic production applications. We selected these applications because they have high network usage and communication and computation patterns that are representative of production applications.

Qbox is a first-principles molecular dynamics simulator [12]. *Crystal Router* demonstrates a many-to-many communication pattern extracted from the Nek5000 application [13]. We removed DUMPI tracing from the *Crystal Router* code so that we can focus only on the communication behavior. *MILC* is an MIMD lattice computation code modeled after applications used to study quantum chromodynamics [14]. *pF3D* simulates laser-plasma interactions [15]. We used a *pF3D* proxy application that emulates only the communication pattern of *pF3D*. To make the application more realistic, we added sleep calls between each communication phase to simulate computation. We tuned the length of the sleep calls so that the communication/computation ratio is roughly 50/50 when *pF3D* is running in isolation[1].

[1] The code for our benchmarks and applications will be released at https://bitbucket.org/lesavoie/rome-2018-network-qos-in-many-core-mpi-applications. Some code will not be released due to distribution restrictions.

C. Environment

We ran our tests on Catalyst, a machine at Lawrence Livermore National Laboratory. Catalyst has 300 compute nodes, each with two 12-core Intel Xeon E5-2695 CPUs and 128 GB of RAM. While Catalyst has tens of cores rather than hundreds of cores per node it gives us the opportunity to study network contention and QoS on real hardware before it becomes a significant concern in future systems. Catalyst's interconnect is a two-level fat-tree built from QLogic InfiniBand QDR hardware, and each node has two network connections [16]. Due to a hardware issue we disabled one of the network connections on each of the nodes. In order to control the amount of network contention experienced by our jobs we ran our tests during dedicated access time when no other jobs were running. This ensures that none of the contention experienced by our jobs is caused by jobs started by other users that we do not control.

We compiled our benchmarks and applications using MVAPICH2 2.2. We used the default version with no changes. As noted in Section II, MVAPICH2 includes the ability to select a service level using an environment variable. In our tests, we used the `IPATH_SL` environment variable to set the service level for each application; this is the environment variable that sets the service level on QLogic hardware. By default, all jobs run in the same service level, so for tests with default QoS we set `IPATH_SL` to the same value for all jobs.

System administrators have set up four service levels on Catalyst. Service levels 0 and 1 have a priority ratio of 9:1. This means that when two jobs are using the same network link at the same time, a job using service level 0 is allocated approximately 9 times as much bandwidth as a job using service level 1. Any job using an uncontested link is allowed to use the full link bandwidth. Similarly, service levels 2 and 3 have a priority ratio of 9:1. However, service levels 0 and 1 are designated "high priority" while service levels 2 and 3 are designated "low priority" with relative priorities of 254:1. This means that, in a contended scenario, service levels 0 and 1 are given 254 times more aggregate bandwidth than service levels 2 and 3. Thus, the effective priorities for the 4 service levels are 2286:254:9:1. Users must have administrative privileges to modify service levels, so we were not able to change these priorities during our testing.

D. Methodology

We evaluated the effect of QoS on our benchmarks and applications by running a large number of tests. Each test involved a group of four applications or benchmarks using either two or four service levels. We used 280 of the 300 nodes to provide a buffer against node failures. We divided the nodes evenly between the jobs, so each job ran on 70 nodes. Our tests included each job running in its allocation in isolation (i.e., with no other jobs running), running all jobs together at the default service level, and running all jobs together with various combinations of service levels. For some tests, we varied the message sizes or computation times of the micro-benchmarks. We ran each test multiple times on different randomized node allocations so that we can investigate the impact of QoS in many different scenarios.

We set up each application such that each run took roughly 60 seconds when running in isolation. In some cases, we did this by modifying the application to quit after a specified number of iterations. While this may cause incomplete computation, it does not affect our network contention analysis. Because the benchmarks do not include set up phases, we set up the benchmarks so that each run took roughly 30 seconds when running in isolation. We restarted each job after it finished until all jobs had completed at least once. This ensures that jobs experience similar network contention throughout their execution time.

Each node on Catalyst has 24 cores across two sockets, and we ran 22 ranks per node. We left one core per socket unused to limit performance perturbation from interrupt processing [17]. We also modified some of the applications to print the elapsed time at the end of each run.

IV. Results

In this section we discuss the results of our experiments with Quality of Service in MPI applications. We include results with different numbers of jobs, different numbers of service levels, different sizes of messages, and different amounts of time spent in computation. We also investigate the underlying reasons for the observed performance.

A. Overall Results

Our first set of experiments is designed to test the overall viability of using QoS to improve throughput on HPC systems. In this set of tests, we ran our four micro-benchmarks simultaneously with each benchmark at a different service level. We tested all possible assignments of jobs to service levels. For the default case we ran all jobs at the same service level. In addition to this, we ran each job with no other jobs running and summed the time from the 4 jobs to produce an "ideal" run time with no inter-job contention. We repeated all of the tests at least 10 times but randomized the allocation of nodes to jobs for each repetition. These results are shown in Figure 1. In this figure, the x-axis shows the assignments of service levels to jobs. The service levels for the four jobs are shown separated by dashes. Jobs are listed in the following order: *Flood-Pairs*, *Nearest-Neighbor*, *All-to-all*, *Random-Pairs*. Thus, the label "0-1-2-3" on the x-axis indicates that *Flood-Pairs* ran at SL 0, *Nearest-Neighbor* ran at SL 1, and so on. As mentioned in Section III, lower numbers indicate higher priority, so in the previous example, *Flood-Pairs* has the highest priority. The y-axis shows the sum of the execution times of each job in seconds. Thus, the y-axis correlates to the throughput because it indicates the amount of time it takes to complete a fixed amount of work. The times are shown as boxplots to show the range of results from the different node allocations. The plots also include dashed lines showing the medians of the ideal and default cases.

This chart demonstrates the motivation behind our work. The penalty for switching from the ideal case to the default

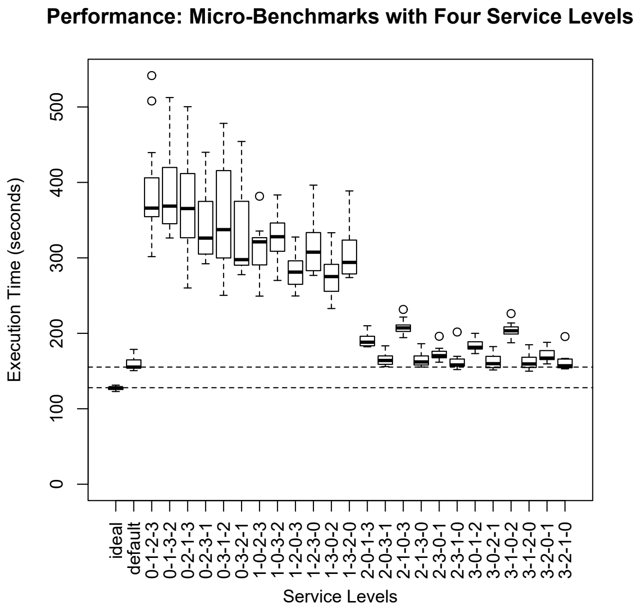


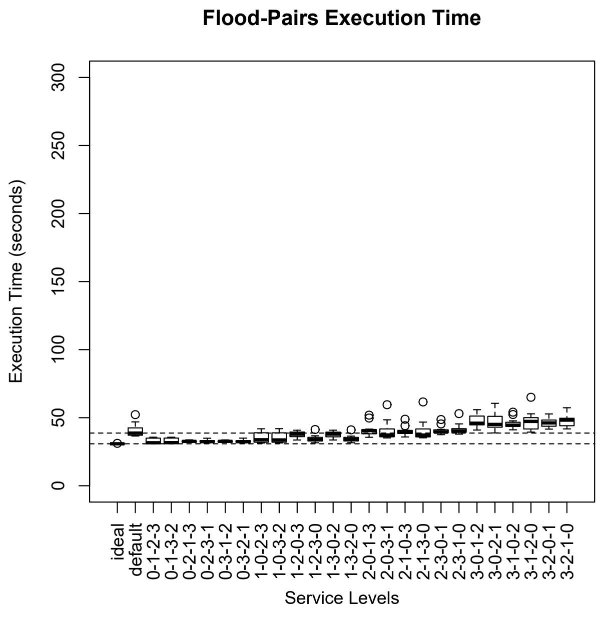
Fig. 1: Overall Performance of Micro-Benchmarks with Four Service Levels

case is around 20% of execution time. Recall that the ideal case emulates an idealized system with no inter-job contention, and the default case runs all jobs at the same service level. The difference between the two is the cost of inter-job network contention. We have additional tests with larger numbers of jobs that show that this penalty can be as high as 70% of execution time. The purpose of our investigation into QoS is to mitigate this penalty.

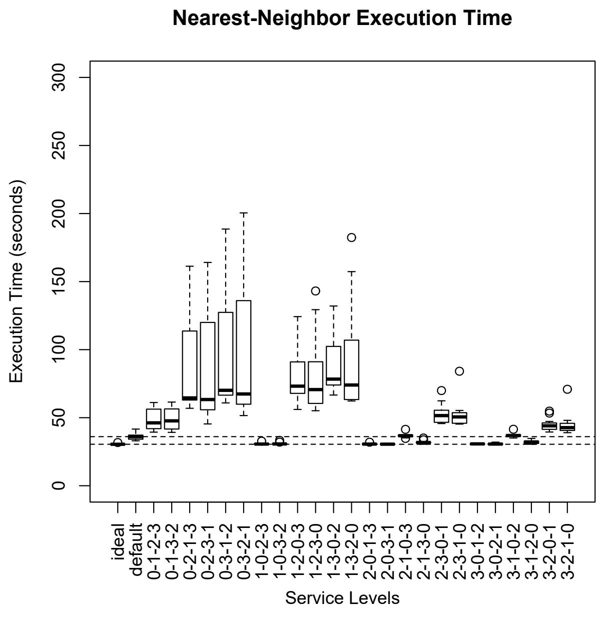
However, our results show that, at least in this configuration, no assignment of jobs to service levels consistently outperforms the default (i.e., all jobs running at service level 0). The penalty for slowing down some jobs completely offsets the benefit of running other jobs at high priority. These results show that per-job QoS can be used to prioritize some jobs above others but not to improve overall throughput. We will analyze this more deeply and discuss the reasons for this lack of improvement in the following sections.

Figure 2 shows the same data broken out by benchmark. Across the benchmarks the execution time approaches the ideal time when it runs at a high service level, but the benchmark is much slower when running at a lower service level. Interestingly, the worst case time is much lower for *Flood-Pairs* than it is for the other micro-benchmarks. This suggests that putting *Flood-Pairs* at a relatively low service level may be the best option with the type of coarse-grained QoS we are testing. Figure 1 bears this out because the tests in which *Flood-Pairs* is at service level 2 or 3 show the best performance. However, this is still not enough to improve overall performance over the default.

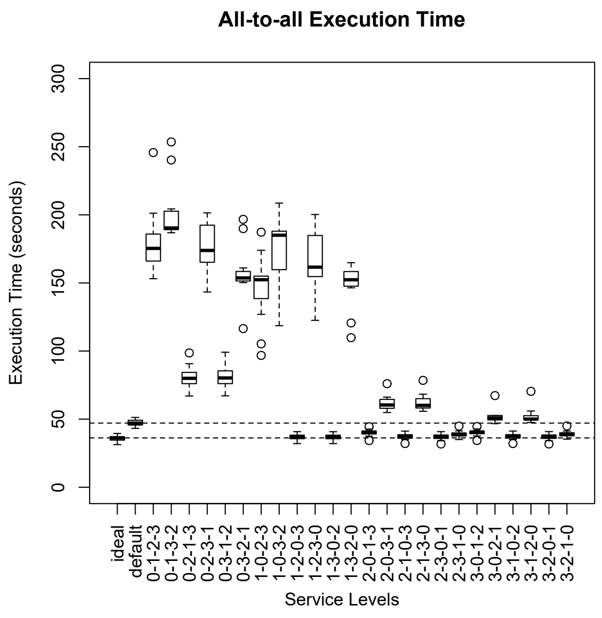
Figure 3 shows the performance of the four applications that we tested. For these tests, the service levels listed on the x-



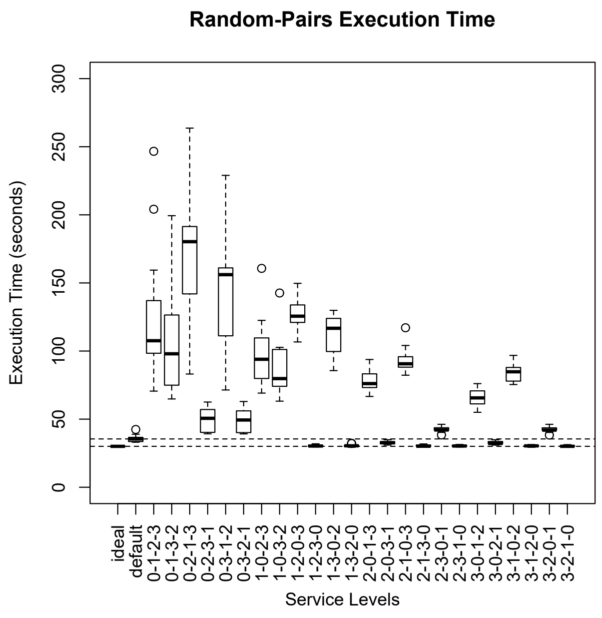
(a) *Flood-Pairs*



(b) *Nearest-Neighbor*



(c) *All-to-all*



(d) *Random-Pairs*

Fig. 2: Individual Performance of Micro-Benchmarks With Four Service Levels

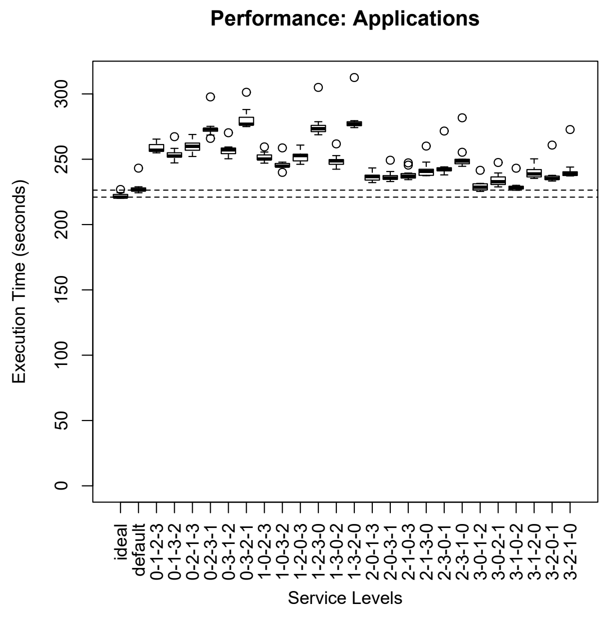


Fig. 3: Overall Performance of Applications with Four Service Levels

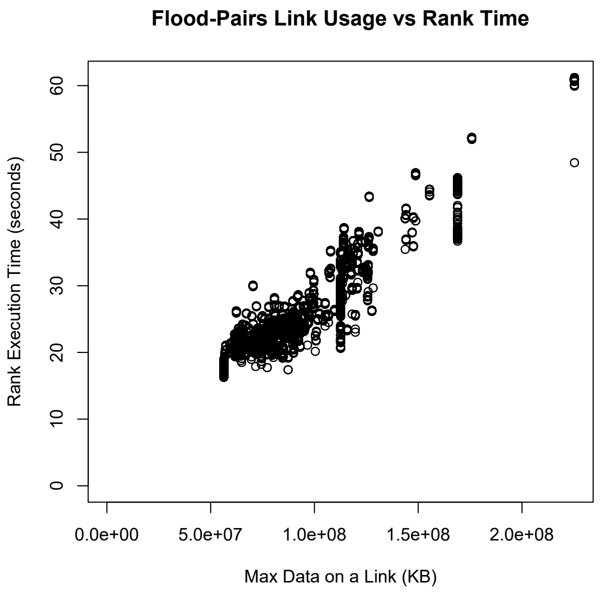


Fig. 4: *Flood-Pairs* Rank Time Compared to Max Data on a Link

axis are assigned to applications in the following order: *Qbox*, *MILC*, *Crystal Router*, *pF3D*. The results are similar to the results in Figure 1 except that the penalty from using multiple service levels is lower. This is because the micro-benchmarks spend 100% of their time in communication, whereas the applications spend between 50% and 80% of their time in communication. Thus, the effect of bandwidth restrictions on the overall run time of the applications is relatively small.

B. *Flood-Pairs Analysis*

In order to better understand the performance of our micro-benchmarks, we examined them individually. We will first discuss *Flood-Pairs* because it is the easiest of our micro-benchmarks to analyze. In *Flood-Pairs*, each rank communicates with only one other rank. Furthermore, the 22 ranks on a given node communicate with the 22 ranks on another node, so performance of ranks on the same node is similar. Therefore, the overall execution time of *Flood-Pairs* is determined entirely by the speed of the slowest pair of ranks.

Figure 4 includes a data point for every rank of every *Flood-Pairs* job we ran in our default tests (i.e., all jobs are running at the same service level). In this figure, tests in which jobs ran at different service levels are not included because that changes the relationship between the amount of data sent on a link and the amount of slowdown experienced by a rank. The x-axis shows the maximum amount of data sent over any link used by a rank. The y-axis shows the execution time of the rank. As expected, there is a correlation between the amount of data sent over a link used by a *Flood-Pairs* rank and the amount of time it takes that rank to execute. This demonstrates that network contention is a major reason for variation in run times across *Flood-Pairs* ranks.

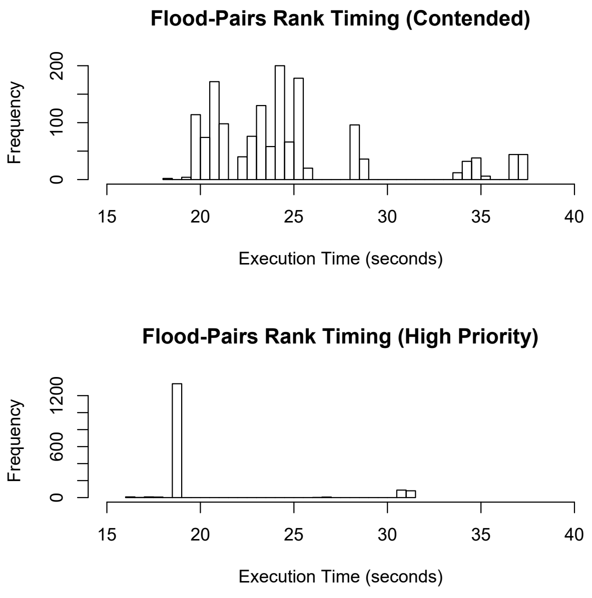


Fig. 5: *Flood-Pairs* Rank Times

We can understand the performance of *Flood-Pairs* more fully by looking at an individual job. Figure 5 shows two histograms of execution times for individual ranks in a single *Flood-Pairs* job running under contention. The histogram at the top shows *Flood-Pairs* rank timing when all jobs are running at the same service level. The histogram at the bottom shows data from the same run except that the *Flood-Pairs* job is given higher priority than the other jobs. These two figures are typical for *Flood-Pairs* jobs from our tests. The overall

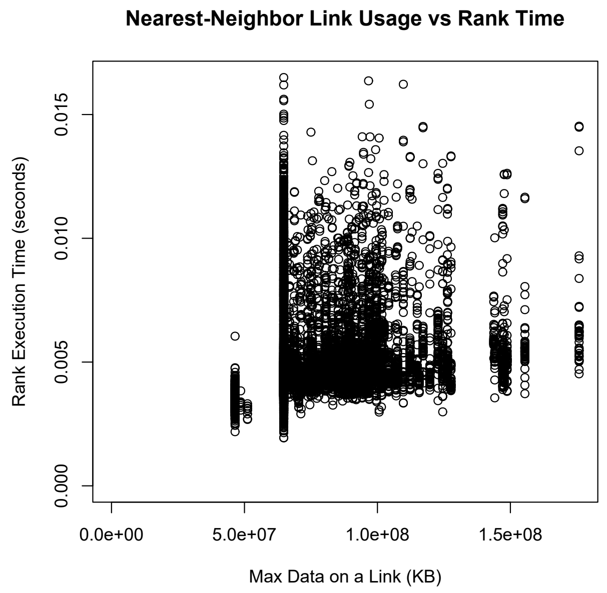


Fig. 6: *Nearest-Neighbor* Rank Time Compared to Max Data on a Link

decrease in execution time from moving *Flood-Pairs* to a high priority service level is relatively small: about 37 seconds to about 31 seconds. This is based on the slowest rank from each job (the rank that is farthest to the right). The execution time of many of the other ranks also decreases; for example, several ranks go from a roughly 28 second run time to a roughly 19 second run time. However, these ranks were already faster than the slowest rank and thus do not contribute to the overall execution time of the job.

The fact that we are decreasing the execution time of ranks that do not affect the overall execution time of the job is significant because it increases the execution time of other jobs. Our data indicates that this is the primary cause of the poor performance we see in Figure 1—coarse-grained QoS decreases the execution time of some ranks unnecessarily, which has negative consequences for other jobs.

C. *Nearest-Neighbor Analysis*

Nearest-Neighbor has a more complicated communication pattern than *Flood-Pairs*. Because each rank communicates with its four neighbors, the effects of slow links can propagate from rank to rank. Thus, a rank can be slowed down by a link that it is not using. This makes *Nearest-Neighbor* more complicated to analyze than *Flood-Pairs*.

Figure 6 demonstrates this problem. This figure shows the runtime of each rank compared to the maximum amount of data sent over any link used by that rank. It is similar to Figure 4 for *Flood-Pairs* except that it only shows the rank times for one iteration right after a barrier. This ensures that the execution time of each rank is only affected by the execution times of its immediate neighbors, not by contention elsewhere in the network. The relationship between contention on a link and rank time is much less direct for *Nearest-Neighbor* than it is for *Flood-Pairs*. In *Nearest-Neighbor*, the contention on a link defines a lower bound for the rank time but not an upper bound. This is expected for a communication pattern in which ranks may be slowed down by other ranks.

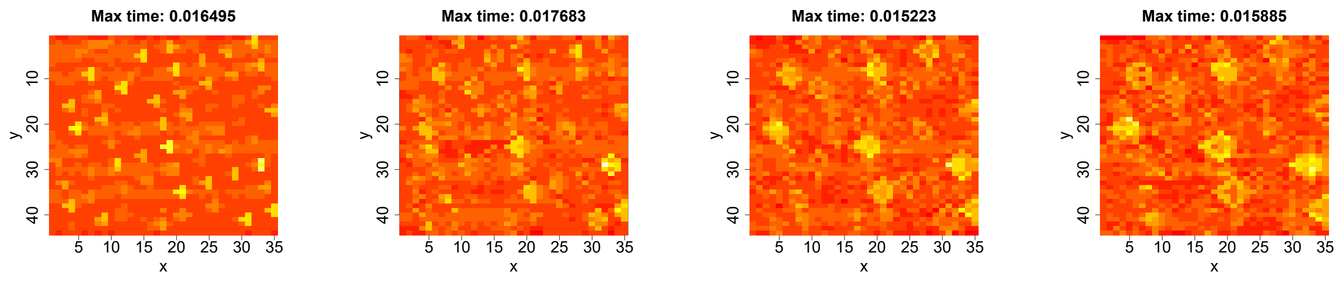
We can analyze individual *Nearest-Neighbor* ranks to obtain some idea of the impact of contention and QoS. Figure 7 shows rank times for several iterations starting immediately after a global barrier. In this figure, *Nearest-Neighbor* is contending with 3 other jobs, all at the default service level. In these images, each rank is represented by a small rectangle. The ranks are arranged in the grid pattern that is used for *Nearest-Neighbor* communication; thus, each rank communicates with its 4 neighbors in the figure. The color of the rank indicates how long it took that rank to complete the iteration; darker colors identify faster ranks and lighter colors identify slower ranks.

Figure 7a, the first iteration after a barrier, shows a regular pattern of slow ranks. The slowest ranks are centered around every 22nd rank, which is the first rank on each node. Because we are running 22 ranks on each node, half of most ranks' communication is on node (i.e., the messages sent and received from the "left" and "right" neighbors). However, the first and last ranks on each node must send and receive 3 out of 4 messages off-node, so these ranks take longer to complete an iteration. In the remaining iterations in Figure 7 we see this contention spread; each slow rank slows down its neighbors in the first iteration, and those ranks slow down their neighbors in the second iteration, and so on. Thus, a few slow ranks can cause rapidly spreading slowdown throughout the application.

Figure 8 shows the same data but with *Nearest-Neighbor* running at a higher priority than the other jobs. We see a similar pattern with the ranks on the edges of nodes taking longer to complete their first iteration after the barrier, which causes neighboring ranks to execute more slowly in subsequent iterations. However, the slowest ranks complete each iteration faster when *Nearest-Neighbor* is at high priority.

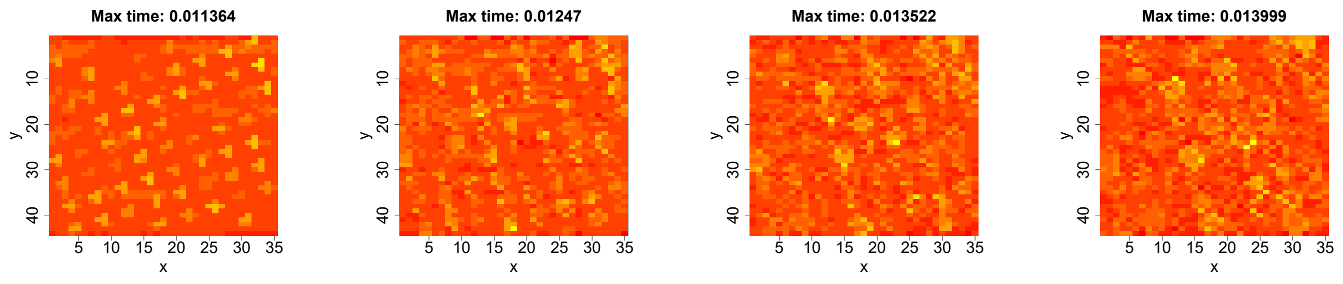
By comparing *Nearest-Neighbor* at default QoS with *Nearest-Neighbor* at high priority we see that some of the disparity between ranks is innate (some ranks spend more time communicating off node than others), but some of it is due to contention. Furthermore, we see that some off-node messages have a greater effect in determining the overall execution time of the benchmark than others, but our coarse-grained QoS scheme forces us to decrease the execution time of all ranks regardless of whether the rank affects the execution time of the whole job. As with *Flood-Pairs*, this increases the execution time of other jobs and offsets the performance gain from QoS.

Figures 7 and 8 also show that a given rank may have a long execution time during one iteration and a shorter execution time during some later iterations. This is unexpected; our analysis indicates that slow ranks should continue to be slow under a static contention pattern. Thus, we believe that this phenomenon occurs because the contention pattern changes. As the execution time of ranks changes across iterations, the times at which ranks send messages also changes, resulting



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4

Fig. 7: Rank Timing Per Iteration for *Nearest-Neighbor* at Default Service Level



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4

Fig. 8: Rank Timing Per Iteration for *Nearest-Neighbor* at High Service Level

in a dynamic contention pattern and unstable rank execution times. This does not change the overall conclusion of our *Nearest-Neighbor* analysis; coarse-grained QoS unnecessarily prioritizes some ranks at the expense of other jobs.

D. *All-to-all, Random-Pairs, and Applications*

Due to space constraints we do not present detailed analyses of *All-to-all*, *Random-Pairs*, and the four applications we used. However, based on our analysis above we expect that these applications exhibit similar behavior to *Flood-Pairs* and *Nearest-Neighbor*. The data we have presented indicates that coarse-grained QoS decreases the execution time of some ranks unnecessarily, which causes slowdowns in other applications and prevents overall performance gains.

E. *Fewer Service Levels*

It is possible that the lack of throughput improvement from QoS is due to the fact that our service levels are so widely spaced. The highest priority application is given 2286 times higher priority than the lowest priority application. In order to investigate this case, we ran additional tests with the four micro-benchmarks but using only two service levels. We tested cases in which the priority ratio was 9:1 and 254:1. Due to time constraints we ran these tests on five random node placements. These results are shown in Figure 9. There are vertical dashed lines between tests with different sets of service levels; the left shows the ideal and default, the middle shows a priority ratio of 9:1, and the right shows a priority ratio of 254:1. Both cases

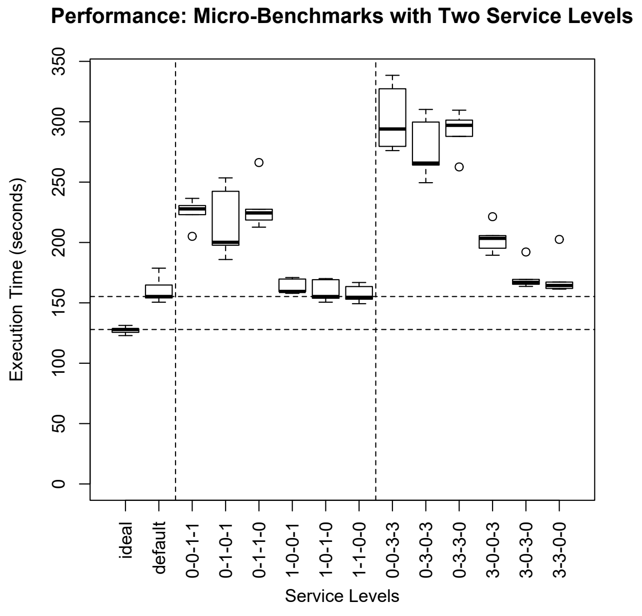


Fig. 9: Overall Performance of Micro-Benchmarks with Two Service Levels

show improved performance over using four service levels, with the priority ratio of 9:1 showing greater improvement than the priority ratio of 254:1. However, QoS still provides

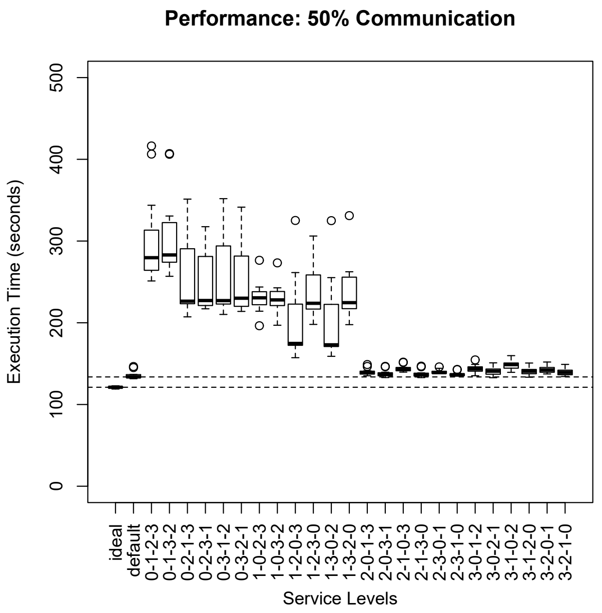


Fig. 10: Overall Performance of Micro-Benchmarks with 50% Communication

little to no benefit over a single default service level. This suggests that the wide disparity in service levels is insufficient to explain the lack of performance improvement.

It is possible that a 9:1 priority ratio is still too high of a disparity between service levels to show improvement over the default case; perhaps levels that are closer to each other will result in better performance than a single default level. Our data, however, provides no evidence that a crossover point exists at which coarse-grained QoS improves performance and throughput, though this possibility deserves more research. At a minimum, our results show that service levels will have to be carefully calibrated to produce any improvement.

F. Varying Computation/Communication Ratio

Our next set of tests considers the interplay between computation/communication ratio and QoS. Figure 10 shows the results of applying QoS when the benchmarks spend roughly 50% of their time in communication. The results are largely similar to the 100% communication results in Figure 1 except that as the amount of communication decreases the impact of contention also decreases. This, in turn, means that the effect of QoS decreases. We obtained a similar trend for experiments in which communication represented 25% of the time, but the results are omitted for space reasons. However, all tests with multiple service levels still perform poorly compared to running all jobs at the default service level.

G. Varying Message Size

We also ran tests with different sized messages. In the small message tests, messages were 16 times smaller than in the tests shown previously. In the large message tests, messages were 4 times larger. Due to time constraints we ran these tests on only two different node allocations. As the message size increases the impact of contention, and thus of QoS, also increases. Otherwise, the results are similar regardless of message size. We omit graphs for reasons of space.

H. Discussion

Throughout our results we found that coarse-grained QoS does not consistently improve job performance and may significantly reduce performance. However, our results suggest that applying QoS at a finer level is likely to mitigate the performance degradation caused by network contention. For example, Figure 5 shows the distribution of execution times of *Flood-Pairs* ranks in the default and high priority cases. Giving high priority to *Flood-Pairs* reduces its execution time from about 37 seconds to about 31 seconds — a 16% improvement. If we assign high priority only to ranks that take longer than 31 seconds to execute in the contended case we can achieve the same improvement by prioritizing roughly 11% of the ranks. This should significantly reduce the impact of QoS on jobs with lower priority. Similarly, *Nearest-Neighbor*'s execution time is determined by only a few ranks. Figures 7 and 8 show that we can reduce the time of the first iteration by 31% by giving high priority to the first and last ranks on each node, which is roughly 9% of the ranks. Similar improvements will show up in the remaining iterations, resulting in an overall performance improvement for the job. Applying rank-level QoS to both jobs at the same time should improve performance for both jobs as long as the high priority messages of the two jobs do not conflict and we are careful not to slow other ranks down to the point that they become the slowest ranks. If we apply similar reasoning to all jobs on an HPC machine we should be able to speed up many of the jobs in a large number of cases. This will improve overall throughput. We will explore these possibilities in our future work.

V. Related Work

The basic ideas behind quality of service have been employed in many areas. In differentiated services [3] (used in the internet), flows are divided into several classes, and each class of traffic can be handled differently. Thus, real time communication might be put into a high priority class while file transfers might be put into a low priority class. We reuse the idea of separating traffic into different classes in this paper. QoS has also been applied in other areas, including video streaming [4], [5], cloud and data center networks [6], and wireless networks [7].

QoS has also been used in HPC to mitigate the problems of network contention. Two papers [9], [10] divide traffic among service levels to reduce head of line (HOL) blocking. HOL blocking occurs when a packet destined for a non-contended link is caught in a switch buffer behind a packet that is destined for a contended link, thus forcing the non-contended packet to wait in the buffer, even though its link is free. Different service levels use different switch buffers, so flows with different service levels cannot experience HOL blocking from each other. However, these papers give all flows the same priority; they do not assign different priorities to different service levels.

A more recent paper [1] by Jokanovic et al. assigns service levels to applications based on their network utilization. Each application is assigned to a separate service level, or if there are more applications than service levels, applications with similar network utilization are assigned to the same service level. Service level priorities are set based on the network utilization of the applications assigned to that service level. In this way, application network interference with other applications is limited, resulting in a performance gain.

The paper by Jokanovic et al. deserves deeper consideration because it comes to the opposite conclusion compared to our work. Jokanovic et al. demonstrated overall throughput improvement with coarse-grained, per-job QoS. In our work we found no such improvement. There are a number of possible reasons for this. First, Jokanovic et al. used simulation, whereas we used hardware. It is possible that the simulation missed some factors that affect network performance. Second, Jokanovic et al. attempted to simulate future hardware while we used current hardware. Third, we used different service levels than Jokanovic et al. We expect that our service levels produce a much wider difference in priority than the ones used by Jokanovic et al. However, our results in Section IV-E indicate that the difference in priority between our service levels was not a significant factor in the lack of performance improvement we saw. In any case, the use of QoS to improve throughput on HPC machines deserves additional research.

Researchers have also considered other methods of dealing with network contention. This includes adaptive routing [18], which routes traffic away from congested links, and job placement strategies that minimize contention [19], [20]. These approaches are complimentary to our work.

VI. Summary

In this paper we have investigated the use of QoS to mitigate network contention by running a number of tests on real hardware. Our tests indicate that coarse-grained QoS, which involves setting service levels at the job level, is unlikely to significantly improve throughput on HPC systems. This differs from previous results based on simulation [1].

Our analysis of job performance with QoS indicates that coarse-grained QoS speeds up ranks that do not need to be sped up. This penalizes other applications and erases the performance gain from QoS. In our future work we will continue investigating QoS in HPC, particularly by applying QoS at a finer-grained level. We expect that this will allow us to demonstrate improved performance from QoS and develop techniques that applications can use to get good performance from future HPC systems.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-745381.

References

[1] A. Jokanovic, J. C. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg, "Effective quality-of-service policy for capacity high-performance computing systems," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, June 2012, pp. 598–607.

[2] E. Krevat, J. G. Castaños, and J. E. Moreira, *Job Scheduling for the BlueGene/L System*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 207–211. [Online]. Available: https://doi.org/10.1007/3-540-45706-2_26

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated service," United States, 1998.

[4] C.-H. Ke, C.-K. Shieh, W.-S. Hwang, and A. Ziviani, "A two markers system for improved mpeg video delivery in a diffserv network," *IEEE Communications Letters*, vol. 9, no. 4, pp. 381–383, April 2005.

[5] W. Kumwilaisak, Y. T. Hou, Q. Zhang, W. Zhu, C. C. J. Kuo, and Y.-Q. Zhang, "A cross-layer quality-of-service mapping architecture for video delivery in wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 10, pp. 1685–1698, Dec 2003.

[6] T. Voith, K. Oberle, and M. Stein, "Quality of service provisioning for distributed data center inter-connectivity enabled by network virtualization," *Future Generation Computer Systems*, vol. 28, no. 3, pp. 554 – 562, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11000392

[7] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar, "Providing quality of service over a shared wireless link," *Comm. Mag.*, vol. 39, no. 2, pp. 150–154, Feb. 2001. [Online]. Available: http://dx.doi.org/10.1109/35.900644

[8] S. A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken, "An overview of QoS capabilities in infiniband, advanced switching interconnect, and ethernet," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 32–38, July 2006.

[9] H. Subramoni, P. Lai, S. Sur, and D. K. D. Panda, "Improving application performance and predictability using multiple virtual lanes in modern multi-core infiniband clusters," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, ser. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 462–471. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2010.54

[10] W. L. Guay, B. Bogdanski, S. A. Reinemo, O. Lysne, and T. Skeie, "vftree - a fat-tree routing algorithm using virtual lanes to alleviate congestion," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 197–208.

[11] (2014, Jun.) Coral benchmark codes. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[12] Qbox code. [Online]. Available: http://qboxcode.org/

[13] Characterization of the doe mini-apps. [Online]. Available: https://portal.nersc.gov/project/CAL/designforward.htm#CrystalRouter

[14] (2016, Apr.) Milc. [Online]. Available: http://www.nersc.gov/research-and-development/apex/apex-benchmarks/milc/

[15] S. H. Langer, A. Bhatele, and C. H. Still, "pf3d simulations of laser-plasma interactions in national ignition facility experiments," *Computing in Science Engineering*, vol. 16, no. 6, pp. 42–50, Nov 2014.

[16] Catalyst. [Online]. Available: https://hpc.llnl.gov/hardware/platforms/catalyst

[17] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03)*, 2003.

[18] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kale, "Maximizing throughput on a dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 336–347. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.33

[19] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! job interference study on dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, 2016.

[20] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: Key to protect job performance predictability," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 449–459.