Mitigating Inter-Job Interference Using Adaptive Flow-Aware Routing

Staci A. Smith*, Clara E. Cromey*, David K. Lowenthal*, Jens Domke‡, Nikhil Jain†, Jayaraman J. Thiagarajan†, Abhinav Bhatele†

*Department of Computer Science, University of Arizona, Tucson, AZ, USA
‡Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, Japan
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA
{smiths949, cerice2, dkl}@cs.arizona.edu, domke.j.aa@m.titech.ac.jp, {nikhil, jayaramanthi1, bhatele}@llnl.gov

***Abstract*—On most high performance computing platforms, concurrently executing jobs share network resources. This sharing can lead to inter-job network interference, which can have a significant impact on the performance of communication-intensive applications. No satisfactory solutions yet exist for mitigating such performance degradation on systems that allow jobs to share the network for the sake of higher utilization. In this paper, we analyze network congestion caused by multi-job workloads on two production systems that use popular network topologies—fat-tree and dragonfly. For each system, we establish a regression model to relate network hotspots to application performance degradation. The models show that current routing strategies are ineffective at balancing network traffic and mitigating interference on production systems. We propose an alternative adaptive routing strategy, which we call adaptive flow-aware routing. We implement our strategy on a fat-tree system, and tests on the system show up to a 46% improvement in job run time when compared to the default routing.**

***Index Terms*—fat-tree, congestion, performance degradation, adaptive routing**

I. Introduction

Scientists developing and optimizing high performance computing (HPC) software usually do not focus on the effects of external interference on the performance of their jobs. This approach works well when optimizing single node performance because compute nodes on most HPC systems are not shared between multiple jobs. However, not all resources are dedicated—in particular, the network and filesystem are shared among all jobs running on a cluster, and this sharing can lead to poor performance. For example, previous performance studies on production systems have found that jobs can suffer significant performance degradation when contending for the shared network [1], [2]. Moreover, it is extremely difficult, if not impossible, for users to develop applications that are fully tolerant of shared resource contention.

HPC systems are expensive investments, both in terms of initial hardware cost and operational cost, and hence the decreased system throughput caused by interference wastes significant time and money. Variation in job performance makes it difficult for users to estimate how much time their jobs will need to run as well as to effectively optimize their codes. This variation also complicates decisions that must be made by system administrators as to how much compute time to award to projects. Thus, mitigating inter-job interference is an important goal for both administrators and scientists using the supercomputer.

One approach to reducing inter-job network interference is to develop routing techniques that minimize hotspots by spreading traffic evenly across the machine. In general, routing strategies fall into two categories: static routing, in which each hop is deterministically decided by an algorithm in advance; and adaptive routing [3], in which current congestion is taken into account in selecting the next hop from several possible alternatives. Static routing algorithms can attempt to load balance traffic—for example, the popular D-mod-k [4] algorithm optimizes load balance under the worst-case assumption of all-to-all traffic. Adaptive routing is also intended to load balance traffic—for example, the scheme proposed by Dally et al. for dragonfly [5] considers congestion at each intermediate router as the message proceeds from source to destination. We use the term *traditional adaptive routing* to describe this type of adaptive routing throughout the paper, to contrast it with the new adaptive approach we develop in this paper.

In practice, however, applications on HPC systems can suffer degradation due to network contention regardless of whether static or traditional adaptive routing is used. Figure 1 shows the performance variability of different applications on two production machines: Cab, a fat-tree cluster at LLNL (left), and Edison, a dragonfly-based supercomputer at NERSC (right). The former uses static routing, and the latter uses traditional adaptive routing. Clearly, neither of these is always able to mitigate congestion arising from network sharing, as run times can inflate by over a factor of two. The adaptive routing mechanism on Edison is sophisticated in that it considers global congestion (in addition to next-hop information) at each intermediate router [5], and it has been engineered by Cray for the Aries interconnect [6]—yet considerable performance degradation still occurs.

In this paper we show conclusively that applications can experience significant performance degradation due to congestion on two modern systems with different styles of routing. On the statically-routed fat-tree system, congestion arises because static routing is oblivious to network traffic state. In

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

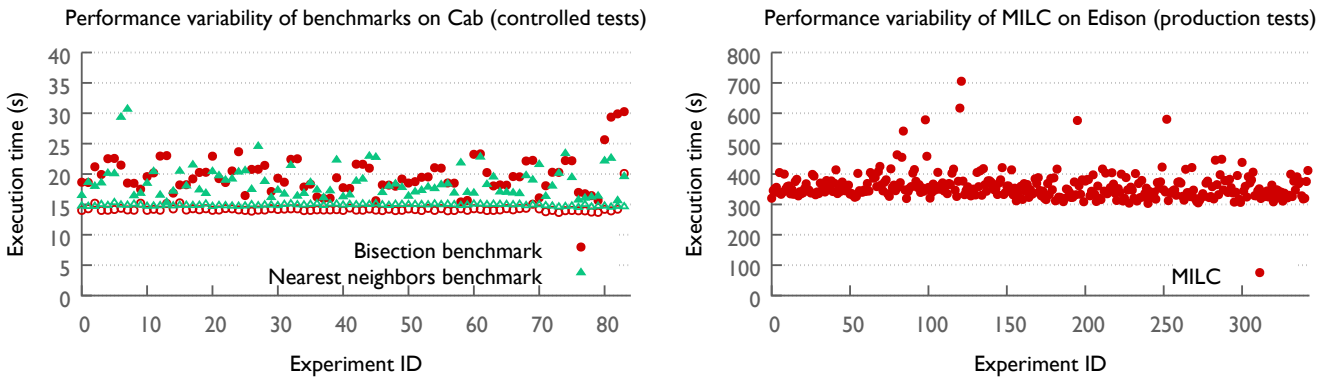

Fig. 1: Inter-job interference can slow down both benchmarks and production applications. Left: Performance of two benchmarks on a fat-tree cluster (Cab at LLNL). The benchmarks spend 75% of their best-case execution time performing computation. Solid points are runs under contention; hollow points are baselines with no contention. Right: Performance of a production application, MILC, on a dragonfly machine (Edison at NERSC).

this work, as an alternative to static routing on fat-trees, we design and implement an algorithm which we call *adaptive flow-aware routing* (AFAR). AFAR uses flow information for each communicating source and destination node pair to re-route data that cross network hotspots. We show that with such global information, much of the slowdown due to congestion can be alleviated. Specifically, we make the following contributions:

1) We present the results of experiments on two large-scale, production machines with InfiniBand-based fat-tree and Cray dragonfly interconnects, respectively. These two topologies are used on many machines in the Top500 list. We find that there are significant slowdowns due to congestion—over a factor of two—on both interconnects.
2) We show that this slowdown is due to network hotspots that are not alleviated by either static or traditional adaptive routing.
3) Based on the observation of hotspots, we develop the AFAR algorithm for the fat-tree interconnect that relieves hotspots using global flow information (unlike traditional adaptive routing or the per-job re-routing proposed in previous work [7]). We implement the algorithm and test it on Cab by modifying OpenSM, the InfiniBand subnet management software.

Results with AFAR show up to 46% improvement in execution time when compared to the default routing algorithm; the median performance improvement across a range of workloads is 25% and 13% for the bisection and nearest neighbors benchmarks from Figure 1, respectively. In general, the performance of all benchmarks in a given workload improves, with the exception of occasional small degradations. While AFAR requires more runtime information than existing routing strategies, these results suggest that it is likely worth the data collection effort, and similar techniques should be explored in more depth in the future.

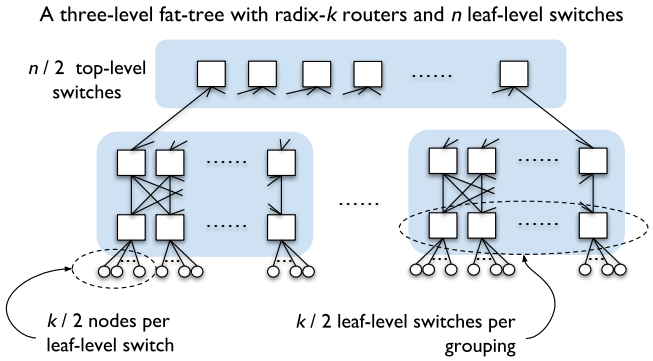

Fig. 2: Connectivity of an example three-level fat-tree.

II. Background

In this section, we discuss the fat-tree topology, the dragonfly topology, and inter-job network interference.

A. Fat-Tree Topology and Static Routing

A popular choice for connecting compute nodes of HPC systems is the fat-tree topology [8]. Conceptually, a fat-tree is a k-ary tree whose bandwidth increases at each level from the leaves of the tree up to the root, so that the links near the root do not form a bottleneck. In practice, however, fat-trees are typically constructed as folded Clos networks, allowing them to be built out of off-the-shelf switches with uniform radix and links with uniform bandwidth.

Here, we describe a typical fat-tree topology that is currently deployed in several HPC systems, illustrated in Figure 2. The fat-tree is constructed using many radix-k switches. At the leaf level, half the ports of each switch connect downwards to $\frac{k}{2}$ compute nodes, and half the ports connect upwards to switches in the next level. Depending on the number of leaf switches, the fat-tree contains either two or three levels for current systems with tens of ports per switch and several

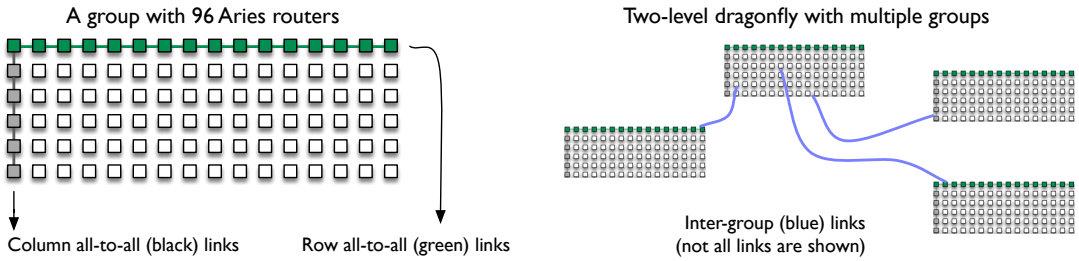

Fig. 3: Example of a Cray Cascade (XC30) installation with four groups and 96 Aries routers per group. Within a group, a message is routed in at most two hops (on the black and/or green links) if congestion does not exist; between groups, the inter-group blue links are used leading to a shortest path of at most five hops.

thousand nodes. In either case, the top-level switches only have downward connections from their ports to other switches (thus if there are n leaf-level switches, only $\frac{n}{2}$ top-level switches are needed).

Traffic in current fat-tree networks is usually forwarded using a static routing algorithm, meaning that all messages between a given pair of nodes take the same (shortest) path through the fat-tree every time. Each path consists of a sequence of links going up from the source node to a nearest common ancestor, followed by a sequence of links going down to the destination node. A commonly-used static routing algorithm is the "destination mod k" or D-mod-k algorithm [4], which load balances routes across links on a fat-tree and is believed to have good performance. In this scheme, the next upward link in the path is chosen at each level based on the destination node's ID, until the common ancestor is reached. After that, downward links that lead to the destination are selected.

B. Dragonfly Topology and Adaptive Routing

The dragonfly topology is becoming another popular choice for interconnection networks in post-petascale supercomputers [5]. In this paper, we focus on Cray Cascade [6] (or Cray XC30), one of the implementations of the dragonfly topology. Figure 3 illustrates a four-group Cray Cascade installation. Ninety-six routers are connected together to form a group, arranged in a 6×16 grid. Sixteen routers in each row are connected in an all-to-all manner by *green* links, and six routers in each column are also connected in an all-to-all configuration by sets of three *black* links per router-pair. Routers in different groups are connected together via *blue* links.

In contrast to fat-trees, the Cray Cascade uses adaptive routing to minimize hotspots [6]. In adaptive routing schemes, each router can dynamically choose between multiple paths for any given message. Some paths are minimal in the number of hops and others go indirectly through a randomly selected third group. Based on the amount of load on the minimal paths, the router may randomly select one of the other non-minimal paths along which to send messages. This random scheme is expected to help mitigate real-time congestion.

C. Inter-Job Network Interference

As mentioned in Section I, jobs in HPC systems typically execute concurrently and contend for shared resources. In this work, we focus on network congestion that arises when jobs compete for the shared system interconnect, degrading communication performance. In certain architectures, for example the IBM Blue Gene machines, jobs are always placed so that they have an isolated partition of the network [9]. However, such placements might lead to system fragmentation and hence lowered system utilization, and most modern machines are configured to let jobs share the interconnect.

The effects of network contention may manifest differently on each machine based on its link bandwidth and topology. In this work we study the effects of network contention on fat-tree and dragonfly machines. While the fat-tree topology has good properties in terms of total available bandwidth across the system, congestion can still be a problem [10]. Under the D-mod-k routing scheme, the next upgoing link in a message's path is selected based on a modulo of its destination ID. Therefore, inter-switch traffic belonging to different jobs may contend at a switch if their destination IDs have the same modulo. In a typical fat-tree installation, multiple many-node jobs are likely to contend for network links and interfere with each other's communication performance.

As mentioned above, dragonfly machines typically use adaptive routing to attempt to load balance traffic, but inter-job network contention can occur regardless. For example, contention can occur for the global links if multiple applications are using non-local communication patterns. Worse, multiple applications can be assigned to the same routers within a group, and even if both have localized (e.g., nearest-neighbor) patterns, they can conflict on row and column links. Outside traffic that is routed indirectly through a given group can also conflict with jobs scheduled to that group on the local links. If the amount of traffic is high enough, congestion will occur in any or all of these locations even with adaptive routing.

III. Experimental Setup

Below, we describe the machines, benchmarks, and production applications used in the experiments for this paper.

A. Descriptions of Machines and Network Counters

Fat-Tree Installation: We ran our fat-tree experiments on Cab, an Intel Sandy Bridge based commodity cluster at Lawrence Livermore National Laboratory (LLNL), connected together using QLogic's QDR InfiniBand network. Cab has 1296 16-core nodes, and is a three-level fat-tree built with radix-36 switches. There are 18 nodes per leaf-level switch, and 72 leaf-level switches on Cab. The link bandwidth on Cab is 40 Gbit/s.

Our fat-tree experiments were run during several Dedicated Application Time (DAT) periods on Cab, meaning that the entire machine was reserved for us and no other users' jobs were running during the experiments. This gave us complete control over all inter-job interference during the runs, and allowed us to select our own job placements. Services such as the Lustre daemon, which run on all compute nodes, were turned off to minimize operating system (OS) noise. We had a total of five DAT sessions over 8 months, with each session lasting either eight, 24, or 48 hours.

Dragonfly Installation: We ran our dragonfly experiments on Edison, which is an Intel Ivy Bridge based Cray XC30 installation at NERSC in Lawrence Berkeley National Laboratory. Edison has 15 groups with 384 nodes each. Since there are more ports available than actually needed for inter-group connections on Edison, six four-link bundles of blue links are used between each pair of groups to provide additional global bandwidth. Edison has a total of 5576 compute nodes, each containing 24 hyperthreaded cores. The total global bandwidth of Edison's dragonfly is 23 TB/s, and its link bandwidth is 4.7 and 5.25 GB/s for global and local links, respectively.

Our dragonfly experiments were run via the batch queue alongside jobs of other Edison users. We ran our target application on the nodes assigned to our jobs by the job scheduler and had no control over inter-job interference. The jobs were always assigned nodes spread across two to 14 groups in our experiments, and our job always shared some group(s) with other jobs. Our dragonfly experiments were executed over the course of six months.

Network Performance Counters: In this work, we show via analysis of network performance counters that hotspots exist and cause performance degradation with current routing algorithms used on production installations. In our fat-tree experiments, we collected global network counters via the subnet management software, OpenSM. The data collected consists of counts of bytes sent on every port in the system every 90 seconds.

In our dragonfly experiments, we periodically collected network performance counters from Edison's Aries routers. The data collected consists of the number of flits (fixed-sized network packets) on each virtual channel of each port as well as the number of stalls incurred while waiting to forward flits from each port. Because a job on Edison only has access to the counters on the router(s) it is attached to, we were only able to collect counters from our application's routers.

B. Descriptions of Benchmarks and Applications

We perform our study with a set of production applications and additional communication benchmarks that we designed. Understanding the performance of a full application can be challenging, so we perform a significant part of our study with the simpler benchmarks. In order to make our study representative of real world scenarios, we designed our benchmarks to be as realistic as possible. In particular, we created our benchmarks to perform communication patterns common in HPC workloads, send reasonable amounts of data, and spend a realistic amount of time in computation.

Each benchmark spends 70-80% of its execution time in computation and 20-30% in communication when run in an interference-free scenario (*in isolation*). This choice is based on profiles of production applications on Cab. Our communication patterns are chosen based on both patterns from our production applications and congestion-prone patterns from prior work [11]. We chose a message size of 64 KB for all the communication benchmarks. Each benchmark is divided into equal-length time steps (composed of several iterations) with a barrier at the end of each step. Within an iteration, the benchmark performs computation and then communication. We chose this structure because it mimics the popular bulk synchronous model for HPC applications [12]. Our benchmark communication patterns are:

Bisection bandwidth (`bis`): Adapted from the ALCF MPI benchmark suite [13], `bis` pairs the first $\frac{N}{2}$ processes with the latter $\frac{N}{2}$ processes, such that process k for $k = 0, ..., \frac{N}{2} - 1$ exchanges messages with process $k + \frac{N}{2}$. Each process repeatedly computes and then exchanges one message with its partner. It is a common pattern considered when testing communication throughput for a system.

Nearest neighbors (`NN`): `NN` lays out the processes in row-major order on a 2D Cartesian grid with configurable number of rows. Each process repeatedly computes and then exchanges a message with each of its four neighbors in the 2D grid. It is a commonly occurring pattern in HPC applications.

Random pairs (`pairs`): `pairs` is similar to `bis` except that processes are paired randomly such that each process has a distinct partner. Each of the first $\frac{N}{2}$ processes repeatedly computes and then sends one message to its partner. In `bis`, all processes on a node communicate with corresponding processes on one other node; but in `pairs`, processes on one node may all communicate with processes on different nodes. This leads to traffic that is distributed across more links in the system than traffic from `bis` and is noted as a difficult-to-handle pattern in the theoretical work of Jyothi et al. [11].

FFT proxy (`fft-proxy`): `fft-proxy` lays out the processes in a 3D Cartesian grid with configurable x, y, and z dimensions. Processes are divided into subcommunicators along the y-axis and repeatedly compute and then perform a single all-to-all communication within each subcommunicator. This mimics one of the communication phases in a production application, pF3D (described next).

TABLE I: Details of benchmarks used in experiments. Computation accounts for about 75% of run time. Message volume is the average data sent per process in each iteration.

Application	Iterations per step	Computation per iter (ms)	Message volume per iter (KB)
`bis`	1800	0.56	64
`NN`	1200	0.83	256
`pairs`	2500	0.40	32
`fft-proxy`	150	10.0	960

We avoid performing actual computation in our benchmarks because it would be susceptible to variability from OS noise and memory contention. Instead, our benchmarks mimic computation via calls to `nanosleep`, which we found to have low variability in our experiments (see the Artifact Evaluation appendix for more details). Table I details the exact configurations of the benchmarks used in the experiments.

In addition to benchmarks, we ran three production applications, namely MILC [14], pF3D [15], and Qbox [16]. MILC stands for MIMD Lattice Computation and is used to study quantum chromodynamics using numerical simulations. We use the MILC application `su3_rmd` [17], which performs a 4D stencil computation, with many point-to-point communications among neighbors and periodic global reductions. pF3D is a communication-heavy laser-plasma interaction code that performs both point-to-point communication and all-to-all communication over subcommunicators in the x and y directions of its 3D Cartesian grid. Qbox is a quantum chemistry application that performs large-sized collective communication and point-to-point messaging over subcommunicators in a 2D Cartesian grid.

We ran experiments with different job sizes varying from 1024 processes (64 nodes) to 6144 processes (384 nodes). All applications and benchmarks were run with 16 MPI processes per node (all cores were used on Cab and 16 out of 24 cores were used on Edison, with no threading). In this study we restrict to running flat MPI programs. The input problem dimensions for each job size are detailed in the Artifact Description appendix.

IV. Analyzing Performance & Network Hotspots

This section focuses on analyzing the performance of benchmarks and applications on both Cab and Edison and relating performance to network hotspots that arise with the current routing.

A. Performance on the Fat-Tree Topology

On Cab, we ran experiments during DAT periods with several different *workloads*, where each workload is defined by the mix of jobs to be run. We study each workload in several different placements, or allocations of nodes to jobs. Specifically, we ran four, eight, and 16-job workloads consisting of either benchmarks or production applications. In this section, we focus on the benchmarks, and the production applications are discussed in Section V.

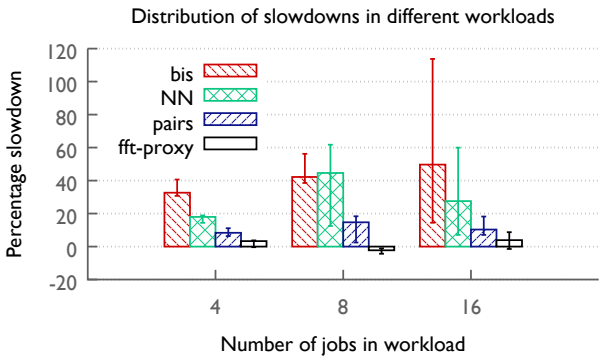

Fig. 4: Distribution of slowdowns (overhead) as percentage of run time in isolation for all benchmarks in three random placements (on Cab). Median slowdown is shown, with error bars for minimum and maximum slowdown.

For our benchmark workloads, we used a total of 1152 of the available 1200 nodes, divided evenly among the jobs. We tried three different randomly-generated placements. In each case, we split the workload evenly between the four benchmarks, e.g., for our 16-job benchmark workload, we ran four `bis`, four `NN`, four `pairs`, and four `fft-proxy` jobs. We used different uniformly-distributed random node assignments for our placements, which is close to the typical case on Cab based on our observation of the job queue.

Figure 4 gives an overview of the slowdown results for all benchmarks at all job counts across the three random placements. Slowdown is calculated with respect to the run time of the benchmark in the same placement when executed in isolation on the machine. The isolated run time is an unrealistic, best-case scenario for a fixed (random) job placement, because no scheduler will leave most of a machine unused just to avoid inter-job interference. In the figure, we see that both `bis` and `NN` are sensitive to inter-job interference, with up to 114% run time increase in the worst case. Additionally, the slowdown for each benchmark varies significantly across placements.

B. Analysis of Fat-Tree Network Counters

Figure 5 shows histograms of link loads across the system for three placements of the 16-job workload. We obtained the link loads from system-wide network counters collected via OpenSM. The load on each link is given in total GB transmitted over the link for the entire duration that the workload was run. Not included in the histogram are links that are not being utilized at all (roughly a third of the available links in the system). Such idle links have been noted previously by Domke et al. and termed “dark fiber” [7].

We can see in the histogram for each of the random placements that the largest bin has links with a load between 15 and 20 GB, and only a handful of links have loads of 60 GB or more. The bins for the highest loads contain just one or two

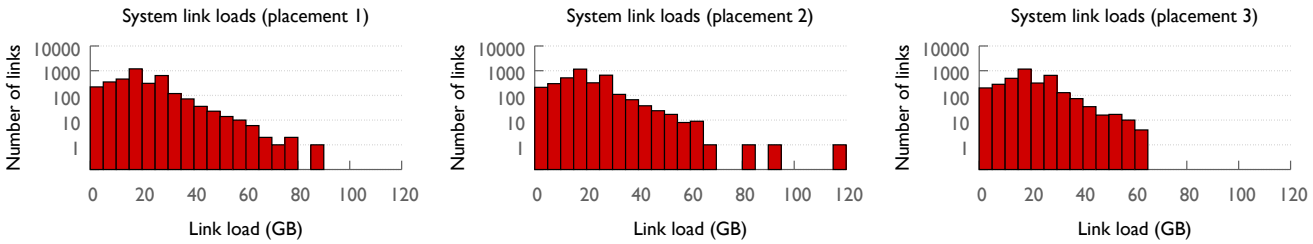

Fig. 5: Histograms of loads on all links of the system for three random placements of a 16-job workload (on Cab). The y-axis is shown in log scale; completely idle links are omitted.

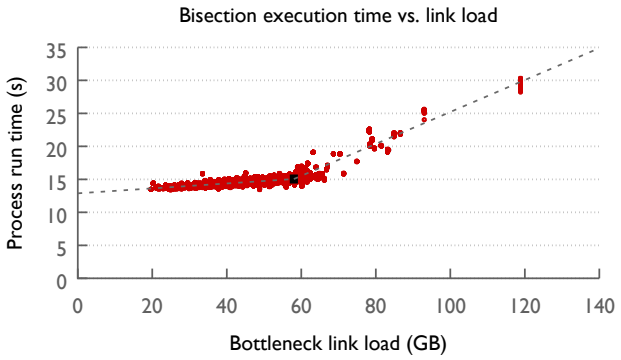

Fig. 6: Time per process versus maximum link load for each process' messages in `bis` in all workloads (on Cab).

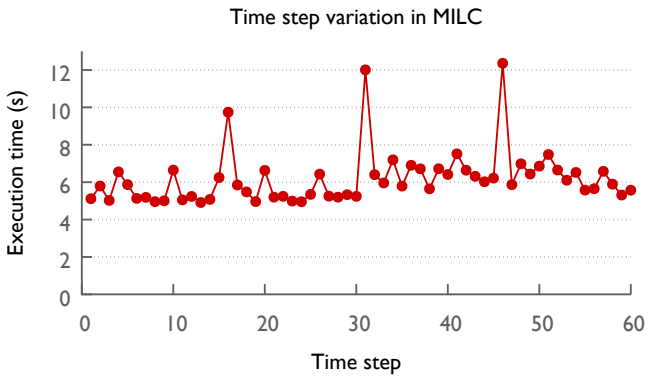

Fig. 7: Time per step for one execution of MILC (on Edison).

links each for the first and second placements; these links are hotspots. The third placement shows no hotspots, and resulted in far less performance degradation than the others. The second placement resulted in the most performance degradation—114%, the largest slowdown seen in Figure 4.

We quantify the load on a given path through the network by considering the most heavily-loaded link in the path—the *bottleneck* link. We then test if the relationship between per-process run time and network load holds across all three random placements by plotting per-process run times in `bis` from every workload versus the load on the bottleneck link for the path taken by messages for that process. The result is shown in Figure 6. We fit a piecewise regression model to this data, finding that the model has an R^2 value of 0.93 with the breakpoint occurring around 58 GB. This point corresponds to a throughput of roughly 3.9 GB/s on average over the course of the workload, which is about 78% of the advertised maximum bandwidth. We conclude that overall job slowdown of `bis` is largely a function of network hotspots that randomly occur with varying placements. Thus, placements that happen to create some heavily loaded links under the static routing scheme (e.g., the first and second placements) result in large slowdowns; those that do not create such hotspots (e.g., the third placement) result in more modest slowdowns.

These results illustrate the fact that network hotspots are to be expected on a statically-routed system, because even well-balanced static routing tables do not take into account the placements and communication patterns of the particular applications running at a given time. The hotspots can cause significant performance degradation and lead to inter-job interference. This suggests that routing needs to adapt to the particular workload on the system. Next, we examine results from a dragonfly system that uses the adaptive routing policy proposed in the work of Dally et al. [5].

C. Performance on the Dragonfly Topology

Unlike on Cab, we could not gain exclusive access to Edison. Therefore, we ran experiments via the batch queue alongside other jobs, using the production application MILC. We ran MILC on 384 nodes with 6144 MPI processes several times over a six-month period and observed its performance while monitoring network hardware counters on routers allocated to our job. Figure 1 shows the execution time of MILC for the different runs. All runs used the same number of nodes and the same input size, yet the variation in performance over different dates is over a factor of two.

Figure 7 shows an execution of MILC broken into its constituent time steps (warmup time steps are omitted for clarity). Each time step of MILC does a uniform amount of work and should take a uniform amount of time, with the exception of every 15th time step, in which MILC does extra computation. However, in many executions, we see highly varying run time per step—performance degradation is occurring during some time steps but not others.

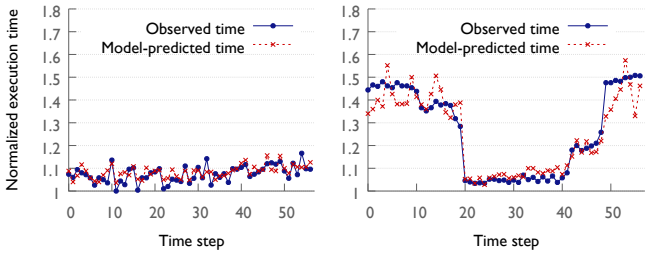

Fig. 8: Predicted execution time for two runs on Edison, normalized to the minimum. A Gradient Boosted Machines model over the sparse features is used for the predictions.

D. Analysis of Dragonfly Network Counters

We collected network counters for each time step of each MILC execution and fit a regression model relating these counters to the variability we observed in MILC's performance. This task is more complicated than it was for the fat-tree for several reasons. First, we only have counters for routers allocated to our MILC job, but MILC's traffic is also sent through outside routers along non-deterministic paths. Second, adaptive routing makes it impossible to separate out the traffic on a port into its constituent jobs. Finally, MILC occupies approximately 100 routers in a typical allocation, with 48 ports per router, so we have counter values for thousands of links over time that must be aggregated without losing critical information.

To help overcome these difficulties, we apply machine learning techniques beyond simple linear regression to our data. In particular, we construct a set of simple statistical features from the flits and stalls counters and build a non-linear regression model with run time as the response variable. Our feature set is comprised of the sum, average and maximum of the counter values for flits and stalls, in each of the eight virtual channels across all allocated routers. We consider each time step in a run as a data sample and construct the feature matrix $\mathbf{X} \in \mathbb{R}^{d \times (N_t \times N_r)}$, where d denotes the number of features, and N_r and N_t correspond to the number of runs and number of time steps in each run, respectively. Denoting y_i as the run time for the sample i, our goal is to build a predictive model, $y_i \approx \mathcal{F}(\mathbf{x}_i)$. Though a variety of regression techniques exist in the machine learning literature, we choose Gradient Boosted Machines (GBM) in our analysis as it has been shown to work well for predicting performance based on network counters [18]. For our GBM, we use the Huber loss function and simple decision trees as the base learner.

Prior to regression fitting, it is common to exploit the low-dimensional structure of the features by applying tools such as Principal Component Analysis (PCA). In addition to regularizing the regression problem, this pre-processing step provides a robust set of features for analysis. However, as a few outlying data points can corrupt the results of PCA, we first perform a matrix decomposition wherein the feature matrix is decomposed as $\mathbf{X} = \mathbf{L} + \mathbf{S}$. Here $\mathbf{L}$ is a low-rank matrix denoting the low-dimensional structure and $\mathbf{S}$ is a sparse matrix describing the outlying data.

To evaluate the fit of our prediction model, we perform 10-fold cross validation. The total set of runs is split into ten partitions, and in each fold, nine randomly chosen partitions are used for training and the remaining one is used for testing. We create three models using the base features $\mathbf{X}$, the low-rank features $\mathbf{L}$, and the sparse features $\mathbf{S}$; interestingly, the sparse features perform best because the outlying components in the counter values are the most useful in predicting the large discrepancies in the execution time. Figure 8 shows the actual run times and the model-predicted run times for two executions from the testing set using the sparse feature model. We can see that GBM makes accurate predictions for most time steps in these executions. The average mean-squared error across folds is 0.35 with a standard deviation of 0.21. This shows that even with partial network counters data, a strong correlation between network counters and application performance exists, indicating that network contention is related to slowdowns.

Despite a sophisticated adaptive routing algorithm, inter-job network interference continues to cause significant performance problems on dragonfly machines. This is despite the fact that Edison uses the vendor-recommended adaptive routing parameters, which have been carefully engineered for the architecture. While different variations of adaptive routing parameters are not explored here, changes to the parameters would almost surely improve performance in some cases but degrade performance in others.

V. Adaptive Flow-Aware Routing

Given our findings in Section IV, we propose an alternative approach to routing, namely using flow information for each communicating source and destination node pair to route adaptively in software. In this section, we demonstrate the feasibility of such an approach by designing and implementing an algorithm that uses centralized, global flow information to re-route data so that network hotspots are avoided. We call this new routing scheme *adaptive flow-aware routing* (AFAR).

A. AFAR Algorithm

We develop AFAR for fat-tree systems because we are able to implement and test it on an existing production system, whereas we cannot control routing on a Cray XC30. We observed in Section IV that significant performance degradation on fat-trees is due to only a handful of hotspots, and we leverage that information to develop a straightforward but effective adaptive routing algorithm for fat-tree networks. A similar algorithm on dragonfly machines is possible in principle, but would be different because the dragonfly topology requires multi-path routing as opposed to single-path routing for best performance [19].

Pseudocode for AFAR is presented in Algorithm 1. The input to the AFAR algorithm is a traffic matrix for the entire workload as well as the routing table. We improve routing by repeatedly finding the hottest link in the system, choosing a destination node that receives data over this link, and re-routing all data sent to this destination node (from any

Algorithm 1: AFAR algorithm for fabric re-routing.

```
   Input: trafficMatrix, rtable, maxLoad, maxIters
   Result: New routing table
1  loads ← getAllLinkLoads(trafficMatrix, rtable)
2  maxLink ← getMaxLink(allLinks, loads)
3  numIters ← 0
4  newRtable ← copy(rtable)
5  while load of maxLink > maxLoad and numIters < maxIters do
       /* Choose arbitrary destination routed across
          maxLink.                                        */
6      dests ← {destinations of data crossing maxLink}
7      D ← first destination in dests
       /* Choose a link to shift load to.                 */
       /* altLinks will never be empty, as it
          contains at least maxLink.                      */
8      altLinks ← {links leading to destination D}
9      minLink ← getMinLink(altLinks, loads)
10     update newRtable entry to reroute data for D to minLink
11     loads ← getAllLinkLoads(trafficMatrix, newRtable)
12     maxLink ← getMaxLink(allLinks, loads)
13     numIters++
14 return newRtable
```

source). The algorithm re-routes the data to the destination so that it crosses an alternative link with the least load. In our implementation, we choose an arbitrary destination node at each step. However, one could easily implement and evaluate different policies for selecting the destination node. The algorithm terminates when a desired maximum link load has been achieved across the entire system.

In the worst case, the algorithm may not improve the maximum link load at every iteration—for example if the maximum load is due to traffic to a single destination, or there is no alternative link leading to the selected destination. In such a case, the algorithm may not achieve the desired link load but will still terminate after `maxIters` iterations. For our prototype implementation, we have intentionally kept the algorithm simple rather than handling possible corner cases. In our tests with realistic benchmarks and production applications, we were always able to significantly reduce maximum link load with this straightforward algorithm.

Currently, we generate the traffic matrix via knowledge of the patterns and volume of communication in our benchmarks, and we apply our algorithm offline. However, a priori knowledge of application behavior is not fundamental to our approach. Hence, we have the option to implement AFAR by using runtime information that could be obtained from within a modified MPI runtime system, a system-provided MPI profiling library, or even the InfiniBand software. The AFAR algorithm could then be invoked periodically in real time, whenever congestion occurs, at the granularity of seconds or minutes.

B. Implementation in OpenSM

We implemented and tested AFAR for a subset of our experiments on Cab. Every InfiniBand network has a Subnet Manager (SM) process which is responsible, among other things, for collecting global data about network state and calculating routing tables for all the switches. The SM on Cab is OpenSM 3.3.19, a standard open-source SM implementation.

TABLE II: Estimated AFAR overhead for production applications if AFAR is applied once a minute (on Cab).

Application	Run time (s)	MPI Calls	Overhead (s)	Percentage
`Qbox`	340	32K	0.26	0.08%
`MILC`	341	913K	0.35	0.10%
`pF3D`	320	144K	0.31	0.10%

OpenSM provides several "routing engines", each of which uses a different algorithm for routing the fabric. On Cab, OpenSM is usually run with the *ftree* routing engine, which implements a type of D-mod-k routing [20].

To achieve the desired routing changes through OpenSM, we performed our routing experiments as follows. First, we obtained current routing tables from the machine shortly before beginning our experiments. For each workload, we input those routing tables into the AFAR algorithm (Algorithm 1), generating improved routing tables. Then, we restarted OpenSM with the *file* routing engine, which loads routing tables from a file rather than computing them by an algorithm. Additionally, we added a new signal handler to OpenSM, which allowed us to trigger OpenSM to reread the routing tables file and update the switches when we wished to modify the routing. In the future, our routing can be implemented directly in OpenSM, rather than via the *file* routing engine.

For each workload, we first measured performance with the original routing tables. When testing the modified routing tables for the workload, we sent a signal to OpenSM to update the switches with the new tables, and waited a short time to allow it to finish. We then observed performance for the workload again.

C. Overheads of Deploying AFAR

To fully deploy AFAR on a system, profiling information must be captured at runtime to determine the current traffic matrix. This will introduce small overheads from intercepting and recording traffic at each node, for example via the PMPI interface, and periodically aggregating this data to send to the OpenSM controller for re-routing. In a complete implementation of AFAR, we estimate the overhead that would be introduced for our production applications based on their number of MPI calls, summarized in Table II.

We address each source of overhead in turn. On Cab, we have found that the average overhead for intercepting and recording the size and destination of traffic is 100-380 nanoseconds per MPI call. The average overhead for reducing the traffic matrix data over a 144-node job is 50 milliseconds. Once the data goes to the OpenSM controller, AFAR itself is run "offline" while normal work proceeds. We have currently implemented AFAR in Python, with no optimization, and have found that it takes a few seconds to run for Cab. Once new routing tables are determined, they must be disseminated to the switches, adding one more source of overhead. In OpenSM, the routing tables are updated at the granularity of *blocks*, with each block containing routing entries for 64 destinations.

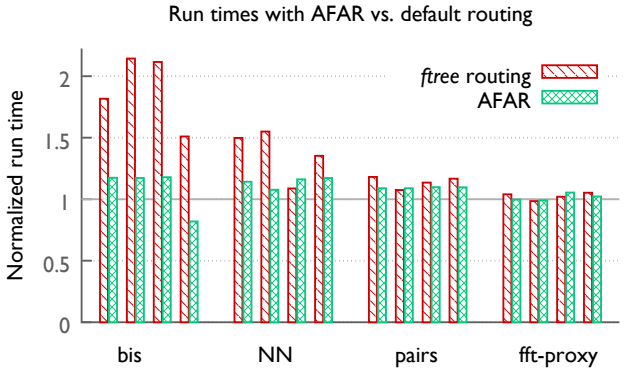

Fig. 9: Run times for each of the 16 jobs in the second random placement, normalized to the run time in isolation (on Cab). Contended run times for the default routing and our modified routing are shown in red and green, respectively.

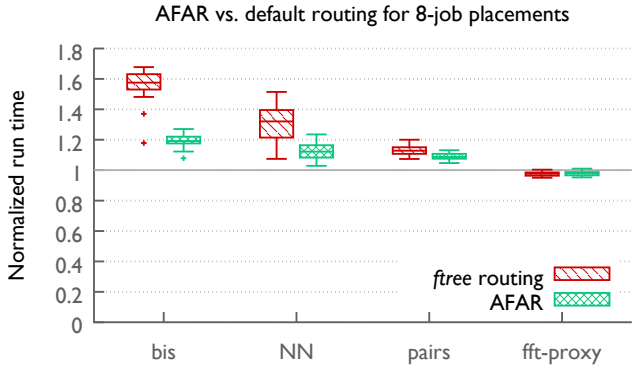

Fig. 10: Distribution of run times for each benchmark across eight placements (on Cab).

During an update, a block is sent out to the switch only if it has changed. OpenSM updates the necessary blocks sequentially. Previous work [7] has measured that the average update time for a single block to a single switch is 4.6 microseconds, though some additional, unexplained overhead is added by OpenSM when updating the system.

With AFAR, OpenSM does not have to update every block of every routing table. It only has to update up to k blocks, where k is the number of routing table entries the algorithm has changed. This property allows our algorithm to scale well to larger systems with several thousand nodes, provided that it can improve performance without updating a large percentage of the routing table entries in the system. In the experiments presented here, AFAR always terminated after only 10 to 20 changes, which has a theoretical cost of less than a millisecond of update time. We found that in reality the total time for OpenSM to update the switches was 3-6 milliseconds.

D. Benchmark Results Using AFAR

In testing AFAR, we focused on eight-job and 16-job workloads. For each of the first and second 16-job workload placements, we tested 15 new routing tables generated by changing one entry up to 15 entries, controlled by the `maxIters` parameter in Algorithm 1. The third placement did not have any hotspots, which the AFAR algorithm detects, leading it to not modify the routing table for that placement. Based on the results with all 15 tables for the second placement, we have selected a good link load threshold for this workload on Cab (about 61.5 GB of total load, or 82% of the maximum link bandwidth). We examine the effects of modified routing on all 16 jobs in the workload for the second placement and the routing table corresponding to this link load threshold. Results for the first placement are similar, so they are not detailed here.

Figure 9 shows the run times for each of the 16 jobs with both the default *ftree* (D-mod-k) routing and with our modified routing. Many of the `bis` and `NN` jobs experienced significantly degraded performance under contention with *ftree* routing, and AFAR significantly improves their performance, returning it to within 17% of isolated run time in most cases. Of the 16 jobs in the workload, 12 perform the same or better with the new routing tables—the median and maximum reductions in run time are 18.6% and 45.7%, respectively. Only four jobs perform slightly worse than with the *ftree* routing, with median and maximum increases in run time of 2.3% and 6.8%, respectively. This shows that the performance degradation due to congestion is not simply shifted from some `bis` and `NN` jobs to the others; rather, we achieve a global improvement for the entire workload through re-routing.

Additionally, we notice that the fourth copy of `bis` actually speeds up from its isolated run time with our modified routing tables. The placement of this `bis` instance caused significant self-contention, elevating the isolated run time from the typical 14 seconds to 20 seconds. The routing modifications eliminate this self-contention, returning the run time to about 16 seconds, despite remaining inter-job interference.

Finally, we also ran eight random placements of eight-job workloads. Rather than trying many routing tables for each placement, we ran the AFAR algorithm until we achieved a maximum link load of 65 GB. Across all eight placements, the maximum number of entries we needed to change to achieve this link load was 10. Figure 10 shows a summary of the results. With this link load threshold for the routing table, we reduce the worst-case slowdown for each placement from between 50% and 70% to between 18% and 27%. In a few instances, one or more of the benchmarks ran slightly slower with AFAR than with *ftree* routing. Across all eight-job experiments, however, the maximum increase in run time under AFAR was 3.2%, compared to a maximum decrease in run time of 29.1%. The median improvement was 4.7% when considering all four benchmarks together. The median improvements for `bis` and `NN`, which are the benchmarks most sensitive to contention, were 25% and 13%, respectively. It is also likely that we could achieve slightly better results by tuning the link load threshold to a slightly lower value.

TABLE III: Run times for each of the eight jobs in the production application workload, normalized to the isolated run time (on Cab).

Application	Time with *ftree* routing			Time with AFAR		
	Job 1	Job 2	Job 3	Job 1	Job 2	Job 3
`Qbox`	1.10	1.08	1.15	1.02	1.01	1.05
`MILC`	1.03	1.04	1.04	1.03	1.03	1.03
`pF3D`	1.03	1.03	(N/A)	1.03	1.03	(N/A)

E. Production Application Results Using AFAR

In addition to running benchmarks on Cab, we ran workloads of four, eight, and 16 production application jobs in random placements. We found that the production applications experienced some unexplained performance variability in addition to that caused by network congestion. However, we saw slowdowns of up to 20% for Qbox and pF3D, and up to 15% for MILC. (The Artifact Evaluation appendix contains details on the repeatability of performance for both production applications and benchmarks in our experiments). Production applications take much longer to run than benchmarks, so we chose one random placement on which to test AFAR. We chose a placement with a high maximum link load and ran AFAR until that load was reduced by roughly 40%.

Table III shows the performance results. We find that the production applications do not interfere with each other as much as the benchmarks do on Cab. This appears to be because the set of applications communicates at a lower intensity than the set of benchmarks, leading to smaller slowdowns in the presence of network contention. However, we find that AFAR still significantly reduces the performance degradation to Qbox without degrading the performance of MILC or pF3d (neither of which degraded much with *ftree* routing).

VI. Related Work

The existing work most closely related to this paper is on scheduling-aware routing and Software-Defined Networking (SDN) in HPC networks. Domke et al. [7] proposed scheduling-aware routing (SAR) for supercomputers, in which routing is periodically updated based on the placements of jobs running in the system. However, SAR is oblivious to the actual communication demands of applications, and hotspots can still occur in the network. In contrast, we use a flow-based approach and leverage information about communicating source-destination pairs in the system.

In SDN, there is work on fat trees [21] that details how InfiniBand can be extended to support OpenFlow-style SDN. Unlike our work, that work requires enhancements to the InfiniBand standard to handle per-flow routes (as opposed to per-destination routes). At present, the scheme is not feasible in a real system, whereas our work is evaluated on a real system. A successor paper [19] presents simulated results of SDN on a dragonfly and compares to adaptive routing.

Routing and congestion control in HPC networks is a broad area of research. Traditional adaptive routing as proposed by Dally et al. [5] is typically used on dragonfly networks. Similar adaptive routing could be used on fat-tree networks, but has not been deployed on such systems historically. Previous work in simulation has found static D-mod-k routing to perform just as well as adaptive routing in fat-trees [4]; but recent tests on Summit, a new fat-tree-based system at Oakridge National Laboratory, showed that the system achieves an additional 26% of the maximum bisection bandwidth when using Mellanox EDR adaptive routing [22]. We have not yet compared AFAR to traditional adaptive routing, but AFAR will have the advantage of global information over a period of several seconds (in a similar spirit to buffered co-scheduling [23]).

Neither traditional adaptive routing nor AFAR handles *endpoint congestion*, which is caused by many-to-one traffic. Complementary solutions for handling endpoint congestion already exist, such as Speculative Reservation Protocol and its extensions [24], [25].

There have been several previous studies of network congestion on various topologies. Bhatele et al. studied performance variation on a 3D torus and determined it was due to network congestion from neighboring jobs [1]. Bhatele et al. and Jain et al. have also studied network performance on two 5D torus (Blue Gene Q) systems, where they could obtain a controlled environment by running experiments in isolation on a midplane of the system [26], [18]. Previous studies on fat-tree have also been performed; for example, Jokanovic et al. used simulation to study inter-job interference of production applications on slimmed fat-trees [27].

There have been publications closely related to our study on dragonfly. The first found variability on a Cray XC30 dragonfly installation Theta [2] due to several different sources in the system, including network contention. The paper notes slowdowns for MILC similar to those we found on Edison. Another paper by Groves et al. correlates network counters to Allreduce performance [28]. Finally, Brandt et al. monitored network counters and showed how congestion forms and abates over time [29]. In simulation, the "bully" paper [30] showed that large jobs can negatively impact smaller jobs.

VII. Conclusion

This paper began by providing evidence that static and traditional adaptive routing on current production supercomputers may be insufficient to mitigate congestion under certain workloads. Our evidence consists of an analysis of network counters, using different regression methods, that shows that network hotspots are possible under both forms of routing and that hotspots correlate well to degraded job performance. Then, we provided details on an adaptive, flow-aware routing algorithm (AFAR) that re-routes traffic to avoid these hotspots. AFAR is implemented via OpenSM on a production fat-tree cluster. When applying the AFAR algorithm, the performance degradation of some jobs reduces from 114% to as little as 17%, when both are compared to isolated run times. While AFAR requires more runtime information than static routing or traditional adaptive routing, we believe this paper makes clear that it is likely well worth the data collection effort.

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-745538). We are also indebted to Livermore Computing at LLNL for their support of our DAT runs.

REFERENCES

[1] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, LLNL-CONF-635776.

[2] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run Variability on Xeon Phi Based Cray XC Systems," in *Supercomputing 2017 (SC'17)*, Denver, CO, USA, November 12-17 2017.

[3] A. Singh, "Load-balanced routing in interconnection networks," Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 2005, http://cva.stanford.edu/publications/2005/thesis_arjuns.pdf.

[4] C. Gomez, F. Gilabert, M. Gomez, P. Lopez, and J. Duato, "Deterministic versus Adaptive Routing in Fat-Trees," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS'07)*, Rome, Italy, March 26-30 2007.

[5] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, pp. 77–88, June 2008.

[6] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: A scalable HPC system based on a dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012.

[7] J. Domke and T. Hoefler, "Scheduling-Aware Routing for Supercomputers," in *Supercomputing 2016 (SC'16)*, Salt Lake City, UT, November 13-18 2016.

[8] C. Leiserson, "Fat-trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, October 1985.

[9] N. R. Adiga, G. Almasi, , Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, and A. A. Bright, "An Overview of the Blue Gene/L Supercomputer," in *Supercomputing 2002 Technical Papers*. Baltimore, Maryland: The Blue Gene/L Team, IBM and Lawrence Livermore National Laboratory, 2002.

[10] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage switches are not crossbars: Effects of static routing in high-performance networks," in *IEEE International Conference on Cluster Computing*, Tsukuba, Japan, September 29-October 1 2008, pp. 116–125.

[11] S. A. Jyothi, A. Singla, P. B. Godfrey, and A. Kolla, "Measuring and Understanding Throughput of Network Topologies," in *Supercomputing 2016 (SC'16)*, Salt Lake City, UT, November 13-18 2016.

[12] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, August 1990.

[13] V. Morozov, J. Meng, V. Vishwanath, J. R. Hammond, K. Kumaran, and M. E. Papka, "ALCF MPI Benchmarks: Understanding Machine-Specific Communication Behavior," in *2012 41st International Conference on Parallel Processing Workshops*, Pittsburgh, PA, USA, September 10-13 2012.

[14] C. Bernard, T. Burch, T. A. DeGrand, C. DeTar, S. Gottlieb, U. M. Heller, J. E. Hetrick, K. Orginos, B. Sugar, and D. Toussaint, "Scaling tests of the improved Kogut-Susskind quark action," *Physical Review D*, no. 61, 2000.

[15] C. H. Still, R. L. Berger, A. B. Langdon, D. E. Hinkel, L. J. Suter, and E. A. Williams, "Filamentation and forward brillouin scatter of entire smoothed and aberrated laser beams," *Physics of Plasmas*, vol. 7, no. 5, pp. 2023–2032, 2000.

[16] F. Gygi, "Architecture of qbox: A scalable first-principles molecular dynamics code," *IBM J. Res. Dev.*, vol. 52, p. 137, 2008.

[17] *NERSC-8 / Trinity Benchmarks*, 2016, http://www.nersc.gov/users/computational-systems/nersc-8-system-cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/.

[18] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, "Identifying the culprits behind network congestion," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '15. IEEE Computer Society, May 2015, LLNL-CONF-663150.

[19] P. Faizian, M. A. Mollah, Z. Tong, X. Yuan, and M. Lang, "A Comparative Study of SDN and Adaptive Routing on Dragonfly Networks," in *Supercomputing 2017 (SC'17)*, Denver, CO, USA, November 12-17 2017.

[20] E. Zahavi, "D-Mod-K routing providing non-blocking traffic for shift permutations on real life fat trees," Technicon Israel Institute of Technology, Haifa, Israel, CCIT Report #776, 2010.

[21] J. Lee, Z. Tong, K. Achalkar, X. Yuan, and M. Lang, "Enhancing InfiniBand with OpenFlow-Style SDN Capability," in *Supercomputing 2016 (SC'16)*, Salt Lake City, UT, November 13-18 2016.

[22] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. H. Oral, D. E. Maxwell, V. G. V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems, and J. Yin, "The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems," in *Supercomputing 2018 (SC'18)*, Dallas, TX, USA, November 11-16 2018.

[23] F. Petrini and W. Feng, "Improved resource utilization with buffered coscheduling," *Parallel Algorithms and Applications*, vol. 16, no. 2, pp. 123–144, 2001.

[24] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally, "Network congestion avoidance through speculative reservation," in *2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA'12. New Orleans, LA, USA: IEEE, 2012.

[25] N. Jiang, L. Dennison, and W. J. Dally, "Network endpoint congestion control for fine-grained communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807600

[26] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, LLNL-CONF-635857.

[27] A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta, "Impact of Inter-application Contention in Current and Future HPC Systems," in *18th Annual Symposium on High Performance Interconnects (HOTI)*, Mountain View, CA, USA, August 18-20 2010.

[28] T. Groves, Y. Gu, and N. J. Wright, "Understanding performance variability on the aries dragonfly network," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Honolulu, HI, USA, Sept 5-8 2017.

[29] J. M. Brandt, E. Froese, A. C. Gentile, L. Kaplan, B. A. Allan, and E. J. Walsh, "Network performance counter monitoring and analysis on the cray xc platform," in *2016 Cray User Group Conference*, London, England, May 8-12 2016.

[30] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch Out for the Bully! Job Interference Study on Dragonfly Network," in *Supercomputing 2016 (SC'16)*, Salt Lake City, UT, November 13-18 2016.

APPENDIX

A. Artifact Description

1) Abstract: This paper includes extensive performance results for applications and benchmarks run under competition for network resources on two different production systems. Details of the systems and production applications are listed here, along with information about where to obtain source code, the complete outputs of the experiments, and the scripts we used for the experiments and analysis.

Our experiments were conducted on production HPC systems and the results of interest are performance results, not computational answers. Therefore, our exact results are not reproducible by a third party, though we make every effort in this appendix to enable a user to repeat the steps we took and observe similar performance results on the systems to which they have access.

2) Check-list (artifact meta information):

- **Program:** MILC, Qbox, pF3D, custom benchmarks
- **Compilation:** instructions given in this appendix
- **Data set:** can be found at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing
- **Run-time environment:** Cab (LLNL), Edison (NERSC)
- **Hardware:** Sandy Bridge, Ivy Bridge, QLogic QDR InfiniBand fat-tree, Cray XC30 dragonfly
- **Run-time state:** Dedicated Application Time, production mode
- **Publicly available?:** code and results are publicly available wherever we have the rights to make them so

3) How software can be obtained (if available): Our paper uses the production applications MILC, Qbox, and pF3D. MILC and Qbox are each available publicly, but the pF3D source code is confidential. MILC can be obtained as part of the NERSC-8 Trinity benchmark suite found at http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/milc/. Qbox can be obtained from Lawrence Livermore National Laboratory (LLNL) at https://github.com/LLNL/qball.

Our paper also uses a suite of benchmarks of our own creation. The benchmarks are available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing.

4) Hardware dependencies: Our experiments were run on two production systems at LLNL and NERSC (the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory). Access to these systems or systems like them is required to reproduce similar results to ours. However, we provide hardware details of the systems for completeness.

a) Cab: Cab is a commodity fat-tree cluster at LLNL. It has 1296 16-core nodes, with 1190 nodes available for batch jobs. Each node contains two sockets, each with one Sandy Bridge processor (Intel Xeon 8-core E5-2670), for a total of 16 cores per node. The processor speed is 2.6 GHz, and each node has 32 GB of memory. The fabric is QLogic's QDR InfiniBand network, connected in a three-level fat-tree with radix-36 switches. There are 18 nodes per leaf-level switch, and 72 leaf-level switches. The link bandwidth on Cab is 40 Gbit/sec.

b) Edison: Edison is a Cray XC30 dragonfly installation at NERSC. Edison has 15 groups with 384 nodes each. Specifically, it is constructed of 30 cabinets, where two cabinets make up a group; each cabinet has 3 chassis; each chassis has 16 compute blades; and each compute blade has 4 dual socket nodes (many more details can be found at http://www.nersc.gov/users/computational-systems/edison/configuration/interconnect). A total of 5576 nodes are available for computing, each containing 24 hyperthreaded cores (two 12-core Ivy Bridge processors). Each node has 64 GB of RAM and three levels of cache, divided into a private L1 and L2 of 64 KB and 256 KB respectively, and a shared 30 MB L3. The total global bandwidth of Edison's dragonfly is 23 TB/s, with an advertised MPI bandwidth of 64 Gbit/sec.

5) Software dependencies: Cab runs the TOSS 2 operating system; Edison compute nodes run a lightweight kernel and run-time environment based on the SuSE Linux Enterprise Server (SLES) Linux distribution. Each system has its own suite of pre-installed compilers and libraries, which were used for this work. Both systems use Slurm for job scheduling, which is required to run our experiment scripts.

TABLE IV: Input problems used for each job size.

Application	Problem dimension
`NN`	$X \times 64$
`fft-proxy`	$8 \times 16 \times Z$
MILC	$64 \times 64 \times 64 \times T$
pF3D	$X \times 16 \times 8$
Qbox	160, 208, 256 atoms

6) Datasets: All details of the input problems used for both production applications and benchmarks are available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing. The complete outputs of our experiments are provided there as well. The basic information is provided in Table IV; `bis` and `pairs`, which have no problem size, were both run with 1152, 2304, and 4608 processes. We used weak scaling for all production applications, so the amount of computational work per process was fixed across sizes. For pF3D, we fixed the y-dimension across problem sizes so that the all-to-all communications are always over subcommunicators of size 16. We ran MILC for four warmup trajectories with five steps per trajectory, plus two work trajectories with 10 steps per trajectory. We ran pF3D for 10 steps, and we ran Qbox for five steps. The Qbox workload was the "gold" input dataset found in the Lawrence Livermore National Laboratory version of the code.

7) Installation: Our custom benchmarks can be built with a simple make command and the mvapich2 `mpicc` compiler. Instructions for downloading and building MILC and Qbox are provided in detail here. All experiments with production applications on Cab used the mpip-mvapich2-1.7 module for mpiP. Experiments on Edison used an mpiP library built from source following the directions at http://mpip.sourceforge.net.

```
# First, download and unzip TrN8MILC7May30.tar. If necessary, rename folder to milc7.
cd milc7
patch -p1 < 0001-Per-timestep-timing-for-2018-re-routing-work.patch
use mvapich2-intel-2.2
cd ks_imp_dyn
make su3_rmd
```

Fig. 11: Commands used to compile MILC on Cab

```
git clone https://github.com/LLNL/qball.git
cd qball
patch -p1 < 0001-Per-timestep-timing-for-2018-re-routing-work.patch
use mvapich2-intel-2.2
autoreconf -i
./configure \
   --with-xerces-prefix=<path-to-spack>/spack/opt/spack/<path-to-xerces-directory> \
   --with-fftw3-prefix=/usr/local/tools/mkl-11.3.2 \
   --with-lapack=<path-to-scalapack>/install/lib/libscalapack.a \
   --with-blacs=<path-to-scalapack>/install/lib/libscalapack.a
make
```

Fig. 12: Commands used to compile Qbox on Cab

a) MILC:

- Code found at: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/milc
- Required patch for per-step timing: 0001-Per-timestep-timing-for-2018-re-routing-work.patch, available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing
- Compiled with mvapich2-intel-2.2 module on Cab, and with our own Makefile on Edison, which is available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing

The commands that we used to compile MILC on Cab are shown in Figure 11.

b) Qbox:

- LLNL version of code found at: https://github.com/LLNL/qball
- Commit number: a33d0438dd83372d645d670bef7e0060-c8933011
- Requires patch for per-step timing: 0001-Per-timestep-timing-for-2018-re-routing-work.patch, available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing
- Compiled with mvapich2-intel-2.2 module
- Build dependencies: Install Xerces through Spack package manager (https://github.com/spack/spack); install Scalapack from scratch (http://www.netlib.org/scalapack)

The commands that we used to compile Qbox on Cab are shown in Figure 12.

c) pF3D: Source code confidential.

8) Experiment workflow: As mentioned, the scripts for running our experiments in production mode (on Edison) and in dedicated mode (on Cab) are available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing.

9) Evaluation and expected result: The scripts we used to analyze our experiment outputs are available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing. The scripts can be run over our outputs to obtain the expected results. Of course, if a third party attempts to re-run our experiments on a different platform or on a different day, the performance results will not be exactly the same as ours.

10) Experiment customization: Our benchmark suite can be adapted to be run on any HPC system that supports MPI, provided a dedicated machine time reservation is obtained. Our analysis scripts can be applied to the outputs to obtain new performance results for the target system.

11) Notes: Our experiments cannot be reproduced exactly due to their nature. However, we have made every attempt to make our results and experiments transparent and to enable other users to run similar experiments with our applications and benchmarks. We also provide an artifact evaluation index with additional details on our data analysis in order to add confidence.

B. Artifact Evaluation

1) Abstract: Our paper includes extensive performance results for applications and benchmarks run under competition for network resources. We take multiple steps to assure the correctness of our benchmarks and limit performance variability due to sources other than the network, including validating the traffic sent by the benchmarks against network counters, using system sleep instead of computation to minimize memory contention and OS noise, and validating the sleep and communication time for each benchmark with timers. We

also increase the reproducibility of our interference results by tightly synchronizing the start times of competing benchmarks.

We also take steps to limit performance variability due to sources other than the network in our production applications, including leaving idle cores to handle OS interrupts and reduce memory contention on node. We evaluate mpiP profiles of the applications after the fact to look for signs of computational performance anomalies and include results only for non-anomalous runs. We have checked the applications for correct execution on our system before reporting any performance results.

2) Dragonfly Results: To avoid computational noise and reduce memory contention in our dragonfly experiments, we used only 16 cores out of the 24 available on each Edison node, and we excluded the first and last cores on each socket (following the advice of Petrini et al. to mitigate OS noise [14]). We have carefully examined mpiP profiles of our application for signs of computational variation between processes and between runs. We have found that our runs all show consistent computation performance, where the computation times of processes range from 197 seconds to 217 seconds, with median 208 seconds. The variance that exists is probably due to factors such as memory contention and turbo. In any case, the computational variance is consistent from run-to-run and much smaller than the communication variance, so we believe that OS noise is not responsible for the observed time variability.

We did not make any changes to the source code of the application, and we validated our compiled MILC executable against a sample problem and its expected output that is provided with the source code.

3) Fat-Tree Results: In our fat-tree experiments, we used four benchmarks, each with a different communication pattern. As described in the paper, each benchmark is divided into equal-length time steps with a barrier at the end of each step. Within a time step, the benchmark performs alternating computation and communication, where the computation is simulated by sleep. Each application is configurable in (1) number of time steps, (2) number of computation-communication phases per time step, (3) length of the computation phases, and (4) size of the messages sent during communication.

In our experiments, we controlled the computation-to-communication ratio of the benchmarks as follows. We fixed the number of time steps to 10. We selected a message size of 64 KB as detailed in the paper, and we tuned the number of communication exchanges in each time step so that a step takes about 0.5 seconds without computation. Finally, for each experiment we set a target computation-to-communication ratio (e.g., 20% of application time in communication, or 4:1) and tuned the computation time to roughly achieve that target. Of course, variability of communication performance from run-to-run causes our ratio to be imperfect, but when run in isolation, our experiments with a target of 25% communication (3:1) always had communication between 20% and 30% for all benchmarks (the percentage inflates when run under competition, as expected).

To avoid actual computation in our benchmarks, which would be susceptible to variability from OS noise and memory contention, our benchmarks mimic computation via calls to `nanosleep`. While it is true that system sleep time is itself slightly variable, we measured total time spent in sleep for our benchmarks across runs and find that it varies by only 0.18% (we used `MPI_Wtime()` as our timer). Because we were able to limit computational noise by using sleep, we do not bother to exclude cores on the node as we did in production experiments on Edison.

The benchmarks were compiled into a single binary, and each workload was run as one MPI program with different subcommunicators performing different communication patterns, which emulates separate jobs. This was done so that the start time of all competing jobs could be synchronized for repeatability of results (see discussion of repeatability below).

As our benchmarks do not perform computation, correctness is defined by each benchmark sending the appropriate-sized messages to the correct partners in between sleep calls. We analytically computed the expected traffic volume for each benchmark and used the routing tables from the machine to discover the expected link loads for each placement and workload. We found that the network counters we collected closely matched the expected link loads; they were slightly higher, which was probably due to traffic during the setup phase of the benchmarks as well as control packets.

Our experiments were run during several DATs on Cab, meaning that we had use of the entire machine, and no other users jobs were running during the experiments. In normal production mode, Cab has access to Lustre, LLNLs high-performance parallel filesystem. However, the Lustre daemon that runs on the nodes can cause significant OS noise in production, so we had the daemon turned off during the reservation.

For each benchmark and placement, we performed three repetitions of the experiment to verify the repeatability of the performance results. With the exception of three anomalous runs out of a total of 504 runs, we did not observe much variation between repetitions of the same experiment. For each job in each experiment, we calculated the standard deviation of its run time across the three repetitions. The median standard deviation for all jobs was 0.05 seconds (where runs took 15-30 seconds). The three anomalous run times were inflated from the minimum by 21%, 77%, and 134%, respectively; but as they represent a small fraction (less than 1%) of our total results, we regard them as isolated incidents. Throughout the description of our results, we report the median run time or median slowdown of the three repetitions, effectively discarding the anomalous outliers.

We also performed three repetitions of each experiment with production applications. In contrast to our benchmark results, we observed larger variance between repetitions of the exact same workload and placement. Across the three repetitions of a given experiment, the maximum slowdown for a given job could be up to 50% higher than the minimum slowdown;

for example, the maximum slowdown was 18% while the minimum slowdown was 12% for one Qbox job. A possible reason for the variance is other sources of performance variability such as OS noise and memory contention between processes of the same job on a given node; these sources are not present in our benchmarks, and our benchmarks showed very small run-to-run variability. Therefore, we focus mainly on our benchmark results in this work, as we have higher confidence in their validity.

For all experiments with AFAR on the machine, we performed three repetitions, and again we found only a small variation in performance between repetitions. The median standard deviation for benchmark run time across the three repetitions was 0.06 seconds (where the run times are all at least 15 seconds), and the maximum standard deviation was 0.31 seconds. For the production application AFAR experiments, the median standard deviation for run time across the three repetitions was only 1.09 seconds (where the run times were all at least 289 seconds), but the maximum standard deviation was 29.7 seconds. We use the median run time for the three repetitions throughout the sections on performance results.

4) Summary: We have described here our efforts to validate the correctness of our applications and benchmarks and to assure the trustworthiness of our performance results (focusing especially on eliminating sources of performance variability from our experiments). In addition, the outputs of all our experiments are publicly available at https://bitbucket.org/stacismith/sc18-adaptive-flow-aware-routing so that they can be independently analyzed by a third party.