# Analyzing Inter-Job Contention in Dragonfly Networks

Staci A. Smith*, David K. Lowenthal*, Abhinav Bhatele†, Jayaraman J. Thiagarajan†,
Peer-Timo Bremer†, Yarden Livnat‡

*Department of Computer Science, The University of Arizona, Tucson, AZ 85721 USA
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA
‡Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, Utah 84112 USA
E-mail: *{smith949, dkl}@cs.arizona.edu, †{bhatele, jayaramanthi1, ptbremer}@llnl.gov, ‡yarden@sci.utah.edu

***Abstract*—Interconnection networks are increasing in importance as node counts increase in high-end machines. To achieve better application performance, newer supercomputers frequently have interconnects with more connections, higher bandwidth, and lower diameter. One example of such an interconnect is a dragonfly topology, which has appeared in multiple recent supercomputers. Adaptive routing and high bandwidth on dragonfly networks leads to the belief that sharing of the network between jobs will not lead to performance degradation.**

**In this paper, we analyze the performance of a production HPC application, MILC, on a dragonfly-based Cray supercomputer. We find that, in fact, the performance of MILC varies by a factor of more than three, and that the performance variation is due to communication delays from network interference. First, we analyze a communication trace of MILC to relate per-rank delays to network activity. Then we use machine learning to develop a predictive model for runtime based on network counters. Our model performs well, with a mean squared prediction error of 0.22.**

***Index Terms*—dragonfly networks, inter-job interference**

## I. Introduction

Optimizing high performance computing (HPC) applications at large scale is a labor-intensive effort, often requiring rewriting code to increase data locality, identifying regions for GPU optimization, overlapping communication and computation, and more. However, applications are usually run in an environment where they must share resources, sometimes causing performance degradation greater in magnitude than the improvement from performance tuning. Examples of such interference include jitter arising from operation system daemons [1], [2], [3], [4] and competition for I/O systems.

Since most applications are communication-bound, an especially problematic type of interference is competition from other jobs using the communication network [5], [6]. Recent network interconnects, such as fat-tree [7] and dragonfly [8], have been designed specifically to remedy such communication contention. Both of these interconnects have a constant number of worst-case hops along shortest paths and combine high bandwidth and adaptive routing to reduce the negative effects of congestion. In theory, these designs allow jobs to share the network obliviously, but in reality this is not the case.

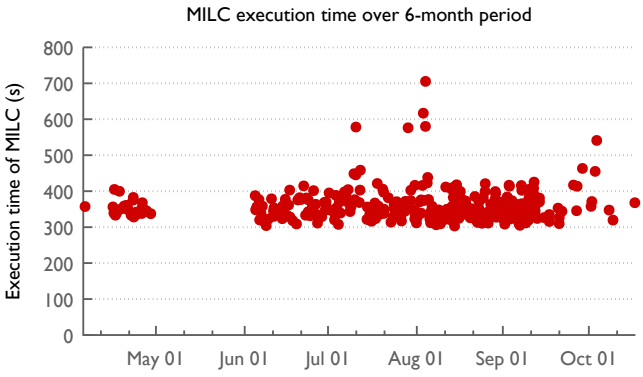


Fig. 1: Execution time for MILC over runs in the batch queue spanning several months.

Consider Figure 1, which shows the execution time of the production HPC application MILC [9] for different runs between April 2016 and October 2016. The runs were executed on a machine with a dragonfly network, NERSC's Edison, in the batch queue. All runs used the same number of nodes and cores per node and the same input size, yet the variance in performance over different dates is large. For the best performing runs, the runtime was between 300 and 320 seconds; during the worst run—on August 4, 2016—the runtime was over 700 seconds.

In this paper, we analyze this variability and find we can attribute it to delay in communication due to heavy network traffic. Specifically, we execute MILC under typical batch job conditions and collect network hardware counters in each timestep to measure network traffic. We use machine learning to develop a predictive model for runtime given these counters, and demonstrate that there is a strong relationship between the amount of traffic on the network and the amount of slowdown that occurs. In addition, we analyze a fine-grain trace of MILC to understand the source of the slowdown and attribute per-rank communication delays to network links experiencing unusually heavy load.

The contributions of this paper are as follows:

- We provide a fine-grain analysis of communication per-

formance in MILC, discovering that certain links are more heavily-loaded despite high bandwidth and adaptive routing, and ranks communicating over these links often experience large communication delays.
- We provide a coarse-grain analysis of overall performance in relation to network activity around a job. In particular, we use machine learning to develop an accurate predictive model for execution time based on network counters, with mean squared error of 0.22.
- We conclude that slowdown in our experiments is due primarily to contention for the network from other jobs, which establishes that the behavior of dragonfly networks is, unfortunately, similar to that of earlier interconnects (e.g., tori) in this regard. Specifically, performance in the presence of network contention degrades by more than a *factor of three* in the worst case, similar to previous results on tori [6].

To the best of our knowledge our work is the first to study inter-job contention on a real dragonfly installation rather than in simulation and to quantify the effect of network contention using hardware counters. Our work suggests that different network designs or communication-aware scheduling algorithms are still needed.

## II. Background

This section covers the necessary background for the rest of the paper. In turn, we discuss variability in computer systems and the dragonfly network.

### A. Variability

Variability in computer systems is an increasingly important topic. With the increasing complexity of chips, software systems, networks, and the environment, variation within and between executions of the same program has become commonplace. This variation causes several problems beyond the obvious waste of valuable supercomputer time. For example, studying performance implications of program optimizations is complicated by variance. From the users' and administrators' perspective, it also complicates estimating the time required for batch jobs and allocation requests for proposals.

One oft-discussed form of variance, especially in high-performance computing, is operating system (OS) noise [3]. With OS noise, a system daemon interrupts at least one core on the system to do its work. From the point of view of the application program, the interrupt time acts exactly as extra computation time. This can have especially negative effects downstream in programs that synchronize at a fine granularity [10], [4]. Several researchers have studied OS noise empirically and in many cases offered solutions (e.g., low-noise operating systems [11]) to reduce or eliminate the effects of OS noise.

However, even with a low-noise OS, there can be many other contributors to variance on a supercomputer. In this paper we primarily focus on the effects of *network variability*, which means that application performance varies because of contention from other jobs using the shared network infrastructure. Network contention can in some cases be eliminated; for example, the Blue Gene approach guarantees each job an isolated piece of the machine with its own dedicated network. The problem with this approach is that it can greatly reduce system utilization. Therefore, the recent trend has been to share the network between jobs on even the highest-end machines.

### B. Dragonfly Network and Adaptive Routing

The dragonfly topology is an architecture that has been proposed to greatly reduce interference when jobs share the network. Because of its believed benefits, it is becoming a popular choice for interconnection networks in post-petascale supercomputers [8]. In this paper, we focus on Cray Cascade [12] (or Cray XC30), one of the implementations of the dragonfly topology.

Figure 2 illustrates the dragonfly topology of the Cray Cascade. The Cray Cascade uses a 48-port network router for connecting nodes in a two-level hierarchy, achieving a low-diameter and high network connectivity. Eight out of the 48 ports are used for connecting four nodes to the router. The remaining 40 ports are used for intra- and inter-group links. There are 96 routers connected together to form a group, arranged in a $16 \times 6$ grid. Sixteen routers in each row are connected in an all-to-all manner by so-called *green* links, and six routers in each column are also connected in an all-to-all configuration by *black* links. The remaining ports are used to connect to routers in other groups via *blue* links.

The Cray Cascade uses adaptive routing to utilize bandwidth. In adaptive routing schemes, each router has multiple paths to choose from for any given message. Some paths are minimal in number of hops and others go indirectly through a third group. Based on the amount of load on the minimal paths, the router may randomly select one of the other paths along which to send messages. This random scheme acts to load balance traffic.

## III. MILC

In order to study the effects of network contention on a dragonfly network in real-world conditions, we select a current production scientific application, MILC. We select MILC because it is load-balanced, performs a common type of communication pattern, and spends a significant amount of time in communication. MILC stands for MIMD Lattice Computation and is used to study quantum chromodynamics using numerical simulations [9]. We use the MILC application, `su3_rmd`, which is distributed as one of the benchmarks in the NERSC-8 Trinity benchmark suite [13].

In `su3_rmd`, MPI processes are arranged in a four-dimensional Cartesian grid for domain decomposition of space-time points. In order to understand the fine-grain trace analysis we present in Section VI, we describe the communication pattern in detail here. A timestep (shown in Algorithm 1) contains alternating computation/communication (nearest-neighbor) phases, followed by a global reduction

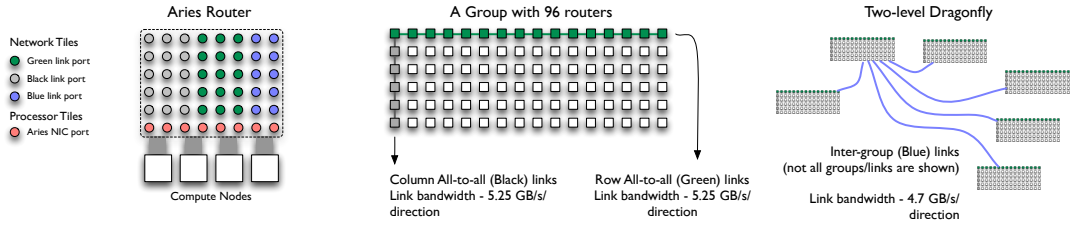


Fig. 2: The dragonfly interconnect.

**Algorithm 1** Overall structure of MILC.

```
1: for t = 1 to NUM_TIMESTEPS do
2:   for i = 1 to 2500 do
3:     First 8-way neighbor exchange + computation
4:     Second 8-way neighbor exchange + computation
5:     MPI_Allreduce(...)
6:   end for
7: end for
```

**Algorithm 2** Communication pattern between Allreduce invocations. For simplicity, only two dimensions (instead of four) are shown and some details of the code are elided.

```
1: MPI_Allreduce(...)
2: for exchange = 1 to 2 do
3:   for size = SMALL, LARGE do
4:     // send negative direction
5:     MPI_Irecv(..., &req1) // +x, size
6:     MPI_Isend(..., &req2) // -x, size
7:     MPI_Irecv(..., &req3) // +y, size
8:     MPI_Isend(..., &req4) // -y, size
9:     computation
10:    // same in positive direction
11:  end for
12:  for req = all requests in order of invocation do
13:    MPI_Wait(req)
14:    computation
15:  end for
16: end for
17: MPI_Allreduce(...)
```

that computes a summation. This pattern is repeated several thousand times and is the dominant cost in MILC.

Details on the computation and communication are shown in Algorithm 2. In the communication pattern, each rank posts a series of `MPI_Irecvs` and `MPI_Isends` to exchange two messages with each of its neighbors. It then calls `MPI_Wait` on all the requests in order. This pattern repeats, and all ranks call `MPI_Allreduce`.

## IV. Experimental Setup

This section describes our experimental setup. In turn we discuss details of Edison, the experiments, and the network counter infrastructure.

### A. Edison Details

In this paper, we focus on Edison, which is a Cray XC30 installation at National Energy Research Scientific Computing Center (NERSC). Edison has 15 groups of 96 routers, with 384 nodes each. There are some site-specific variations to the Cray Cascade topology on Edison. Since there are more ports available than actually needed for this small installation, three black links are used to connect each pair of routers in the intra-group columns. In addition, six four-link bundles of blue links are used between each pair of groups.

Edison has a total of 5576 compute nodes, each containing 24 hyperthreaded cores (two 12-core Ivy Bridge processors). The remaining nodes are reserved for login and I/O. Each node has 64 GB of RAM and three levels of cache, divided into a private L1 and L2 of 64KB and 256KB respectively, and a shared, 30 MB L3. The total global bandwidth of Edison's dragonfly is 23 TB/s global bandwidth.

There are four adaptive routing algorithms available on XC30 systems; in our tests, the default routing algorithm on Edison is used.

### B. Experiment Details

In this paper, we ran the experiments as follows. We ran MILC as a flat MPI program with 6144 processes, spread over 384 nodes and 16 cores per node. The input size we used was $64\times64\times64\times96$, with 80 timesteps total. The "warmup" timesteps used four trajectories and five steps per trajectory, while the "regular" timesteps used four trajectories and 15 steps per trajectory.

We always requested 384 nodes for our job to execute because it is the exact size of one group. We ran the application over whatever nodes we were assigned. This means that our job could run on anywhere between one group and all groups, though in practice we were always assigned between two and 14 groups in our experiments. The nodes that our job received competed with another job in at least one group in every one of our experiments.

### C. Network Counter Details

In our experiments, we periodically collected network hardware counters from Edison's routers. Because an application on Edison only has access to the counters recorded on the router(s) it is attached to, we were only able to collect counters from MILC's routers for our analyses.

Specifically, we collected flits (the smallest-sized packets sent on the network) and stalls (incurred on flits when they

cannot be forwarded from the router). Note that the larger messages that applications send are broken up into flits to be transmitted over the network and that all flits are of uniform size. On Edison, the counters are collected at every router on a per-link basis. The flits counter for a link records how many incoming flits were received on that link, and the stalls counter is incremented every time a flit received on that link incurs a stall while waiting to be forwarded. Together, these metrics give an idea of the congestion on a link.

## V. Overall Results

This section provides overall results of our MILC experiments. Specifically, we discuss the overall variability in computation and communication time across runs of MILC and discuss the impact of how nodes are placed onto groups on Edison.

### A. Overall Variability in MILC

We provide a general breakdown of computation and communication times for MILC in Figure 3. The left-hand graph shows an error-bar plot of computation and communication times across ranks in our runs of MILC. Generally, this graph shows that best-case runs of MILC have about a 60-40 split between computation and communication time.

To avoid computational noise in our 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. [3]). Our runs all show consistent computation performance, where the computation times of ranks range from 197 seconds to 217 seconds, with median 208 seconds. From this we conclude that OS noise is not responsible for the observed time variability. Since computation time is consistent across our runs, the variation in total runtime is due to communication, as can be seen in Figure 3.

The right-hand graph shows the contribution of the dominant MPI calls in MILC. Here, it is clear that two calls represent roughly 80% of the communication time: `MPI_Wait` and `MPI_Allreduce`. This is because MILC uses non-blocking operations for the point-to-point messages, so most of the communication time is accumulated in calls to `MPI_Wait` and `MPI_Allreduce`. The leftmost bar of the graph shows the breakdown for a run of MILC which had an elevated execution time. In this case, `MPI_Wait` and `MPI_Allreduce` calls account for almost 95% of the time spent in MPI.

### B. Impact of Group Placement

We briefly consider the possibility that for a fixed number of nodes, an increase in the number of groups over which those nodes are allocated may increase execution time. On other architectures (e.g., tori) contiguity of placement has an impact on performance. In theory, placement on a dragonfly should have little impact since minimal paths are at most five hops, and the inter-group bandwidth is high.

Figure 4 shows the variation in performance as a function of the number of groups. It appears that the number of groups has no direct impact on performance. We observe that scheduling policies that attempt to minimize number of groups are therefore not sufficient to avoid performance degradation.

## VI. Fine-Grain Analysis

This section provides analysis of a detailed MPI communication trace of the application to determine the likely cause of communication variation.

### A. Analysis of `MPI_Wait` Times

In our experiments, we observed a wide range of execution times for MILC; Figure 4 shows the overall variation in run time. We see that the worst-case MILC execution time is a factor of 2.2 slower than the best-case time. At the scale of individual timesteps, the factor is 3.6, which serves as an upper bound for overall slowdown in our experiments.

As mentioned above, the slow execution is due to extra communication time, specifically in `MPI_Wait` (and `MPI_Allreduce` by extension). In order to better understand what happens across ranks at individual calls to `MPI_Wait`, we took detailed traces of the pattern shown in Algorithm 2 (see Section III) in several executions of the program. Because MILC makes so many communication calls in an execution, the memory overhead of capturing a complete trace is prohibitive. Therefore, we kept track of traces for the longest-running and shortest-running reductions in each execution and saved only those traces. In the seven executions we traced, the shortest reduction time was always less than a millisecond, and the longest reduction time varied from 5-71 milliseconds. We examine the period between reductions for the worst-case (71 millisecond) reduction and the corresponding best-case reduction in the same run.

First, we find that in the fast period, the `MPI_Wait` times for all messages are well below 1 millisecond. In the slow period, many ranks have `MPI_Wait` times above this threshold, with the worst case `MPI_Wait` time almost 35 milliseconds. Figure 5 shows the times per rank at an example `MPI_Wait` call in the code.

An inflated `MPI_Wait` time at rank A could be caused by one of two things: (1) the partner rank for A posted the matching MPI operation after A arrived at the `MPI_Wait`, or (2) the partner rank posted the matching MPI operation before A arrived at the `MPI_Wait`, but the communication itself was slow. Unfortunately, we cannot order events on different nodes relative to each other, so we cannot determine directly whether a partner posted its `MPI_Isend` or `MPI_Irecv` on time. However, because the size of the second message to each neighbor is 18KB, the rendezvous protocol is used for the `MPI_Isend`. Therefore a rank may block when `MPI_Wait` is invoked on a previous `MPI_Isend` as well as a previous `MPI_Irecv`, and we determine that a delay falls into case (2) if both the sender and receiver of the given message block (see Figure 6).

Recall that in our experiments we used 16 MPI ranks per node, typically with 4 nodes per router. Therefore some communication occurs between ranks located on the same

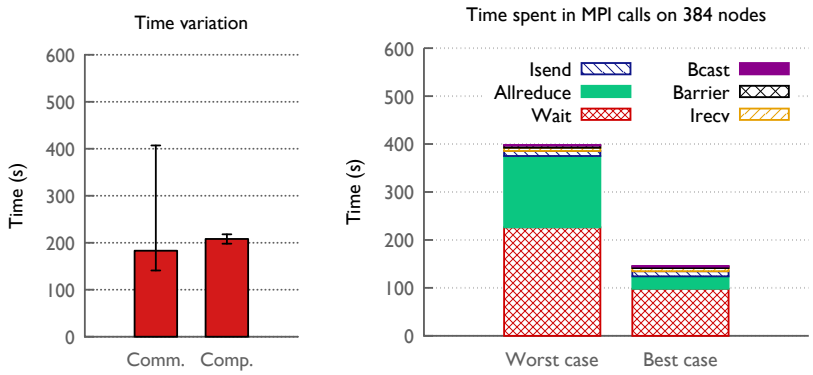


Fig. 3: Breakdown of computation and communication times. On the left is overall time spent on computation and communication per rank in our experiments. On the right is time in the most commonly invoked MPI calls.

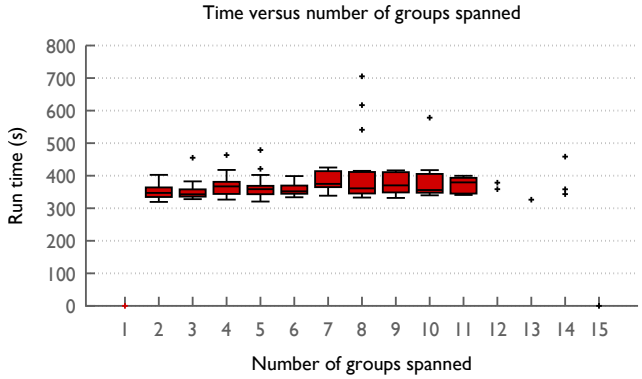


Fig. 4: Variation in performance given the number of groups.

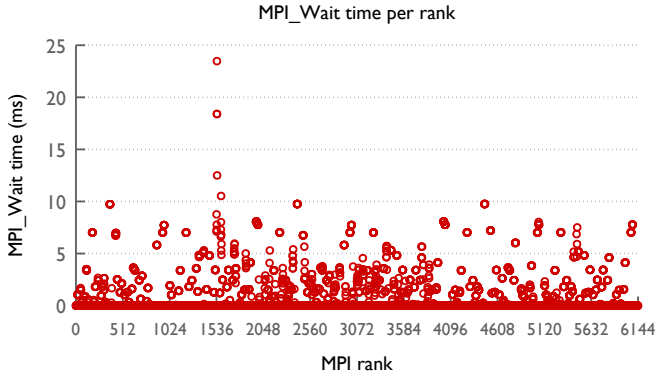


Fig. 5: Example `MPI_Wait` times per rank at one neighbor exchange of our trace.

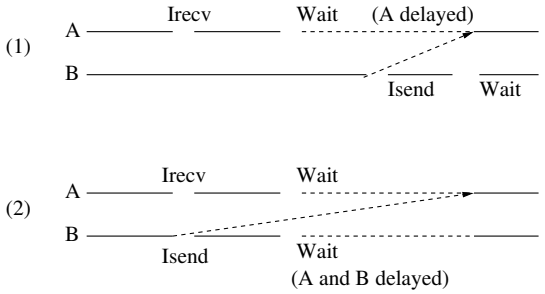


Fig. 6: Two possible reasons for delay on an `MPI_Wait`.

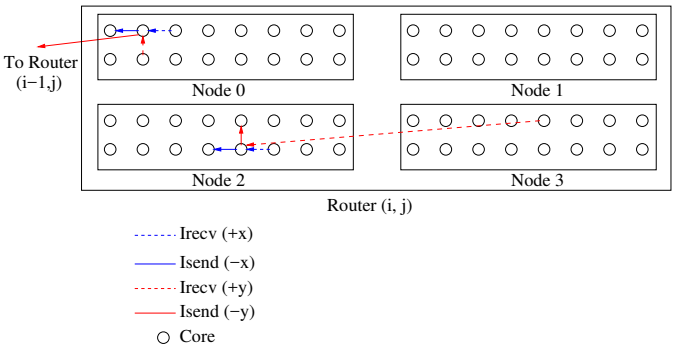


Fig. 7: MILC communication pattern, as mapped to nodes.

node or on nodes connected to a common router. Due to the way the problem grid is mapped to ranks, most neighbors in the $x$-dimension fall on the same node (see the blue arrows in Figure 7). As expected, we see no delays at the first message exchange in this dimension. However, many neighbors in the $y$-dimension, and all neighbors in the $z$- and $t$-dimensions, fall on different nodes (the red arrows in Figure 7 illustrate the $y$-dimension neighbors). At the first message exchanges in these dimensions we see `MPI_Wait` times up to 35 milliseconds. Table I gives a breakdown of delays due to a partner arriving late (one-sided delays) and delays due to slow communication (two-sided delays).

We find that many ranks experience one-sided delays. Some are due to the partner rank absorbing the delay at a previous `MPI_Wait`. Others must be due to the partner rank arriving late, as discussed above. Because all ranks spend a consistent amount of time in computation up to that point in the trace, those partner ranks must have been late in leaving the previous

MPI_Allreduce. Because MILC experienced significant communication slowdowns on the previous few reductions, it is likely that communication in the MPI_Allreduce itself was delayed, causing some ranks to exit late. In any case, we also see many delays due to slow communication between neighbors (these include the largest delays), and we explore these further by analyzing performance counter data from the network during the trace period.

| Neighbor type | Delay type | Delayed pairs | Max delay (ms) |
|---|---|---|---|
| On node<br>3072 pairs | | 0 | |
| On router<br>1968 pairs | Late partner<br>Slow comm. | 1536<br>0 | 9.76 |
| Same group<br>11,040 pairs | Late partner<br>Slow comm. | 3203<br>239 | 9.74<br>7.5 |
| Different groups<br>2352 pairs | Late partner<br>Slow comm. | 382<br>736 | 21.88<br>34.78 |

TABLE I: Summary of delays at the first $y$-, $z$-, and $t$-dimension exchanges. There are a total of 18,432 (6144×3) ordered pairs of communicating ranks. Times for two-sided delays are reported as the minimum of the receiver's delay and the sender's delay on a given message.

### B. Analysis of Network Counters

Figure 8 shows the relationship between MPI_Wait times in the first off-node message exchanges and the network counters for the links between the communicating ranks. Because (1) we only have the network counters for MILC's routers, and (2) the paths along which messages are routed are non-deterministic (because of adaptive routing), we do not have global data and cannot locate exactly which links are used for the messages sent by our MILC application. However, many of the off-node neighbors lie in the same row or column of the same group and therefore have a direct link between them. In the absence of contention for that link, messages should always be routed across it, which simplifies our analysis. Therefore, in the figure we show only those pairs of ranks that lie in the same row or column. Omitted here is a graph of MPI_Wait times vs. total number of flits on the rank's router; it looks similar but the shape is less pronounced.

Clearly, there is not a linear relationship between MPI_Wait time and the number of flits on the network. However, below a certain threshold of flits on the connecting link, MPI_Wait times are always small (near zero). Above that threshold, there is an increasing probability that a rank will be delayed. Evidently when there is competition for the network near a rank (as indicated by higher-than-usual flits), that rank may experience a significant delay while its messages are delivered. We see many such delays in this trace. There are several possible reasons why some other ranks do not experience delays despite having nearby network competition:

1) The adaptive routing scheme successfully routes the communication along a less congested path.

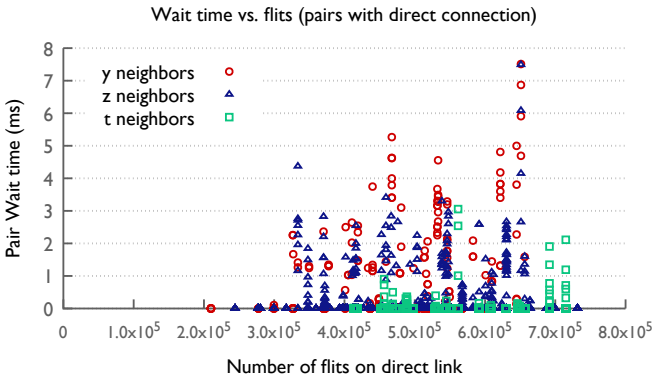


Fig. 8: MPI_Wait time at the first set of off-node neighbor receives vs. flits on the connecting row or column link. Only ranks with a direct link to each other are shown, and ranks with late partners are omitted.

2) The other application(s) competing for the network have distinct computation and communication phases, and these application(s) are in a computation phase.
3) The messages of the other application(s) competing for the network do not happen to collide with the fast rank's messages.

It is possible that all of these occur here. Evidence of numbers 2 and 3 is discussed more below.

We omit figures of MPI_Wait time vs. stalls because they do not show a clear relationship. One reason is that the stall counter that we collect does not indicate which outgoing port(s) cause the stalls. Another is that we have no way to determine if the stalls are incurred on MILC's traffic or other jobs' traffic. In addition, high stalls on a link trigger the adaptive routing to choose alternate (indirect) paths, making the expected relationship between stalls and time unclear.

For reasons of space, we do not analyze other traces in detail here. However, the execution with the second-longest reduction showed very similar results to the one discussed above.

Finally, we consider the counters for our other traces over the entire period between two consecutive reductions. Figure 9 shows the distributions of flits across all links for both the best-case and worst-case such periods, over seven execution traces. Across runs, there is not a linear relationship between total flits and runtime, which suggests that the rate of background traffic not attributable to MILC is highly varying over time. However, a general relationship between total flits and runtime holds with the exception of one anomalous run (the second one). The 57-millisecond period is due to a single long delay on one pair of nodes in the second 8-way neighbor exchange. Because the nodes are attached to the same router, we believe that an anomalous event happened on this router or one of the nodes. We cannot account for the three-millisecond period with inflated flits, but we did not see such behavior in any of our other traces. We believe that the high flits are due to competing job(s) sending voluminous traffic over the network

during a short time when MILC was not communicating. However, we are not able to obtain information at a fine enough granularity to verify this.

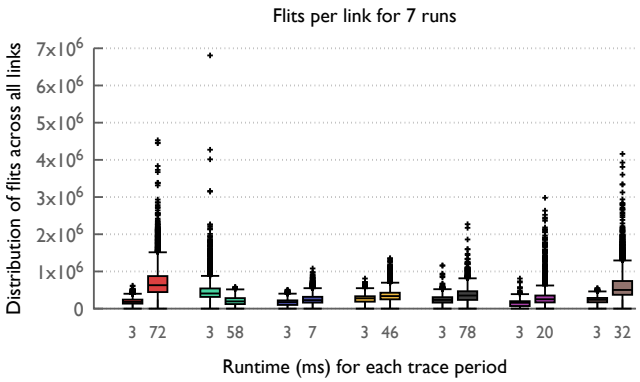


Fig. 9: Distributions of flits across all links for minimum and maximum reduction traces in seven runs. Each pair of bars shows the two traces from a given run.

## VII. Coarse-Grain Analysis

In this section, we employ machine learning techniques for a coarse-grain analysis. As seen above, there is a relationship between per-rank communication time and network counter data within a single communication period of MILC. Here, we demonstrate that a relationship exists between execution time and network counter data across all executions of the program.

### A. Predictive Modeling

Figure 10 shows an execution of MILC broken into its constituent timesteps (warmup timesteps are omitted for clarity). Each timestep of MILC does a uniform amount of work and should take a uniform amount of time, with the exception of every fifteenth step, in which MILC does extra computation. However, in many executions we see highly varying runtime per timestep. Since it is impractical to capture sufficiently fine-grain counter and MPI data for an entire execution, we capture counters at the end of each timestep. Our goal is to fit a model that predicts timestep runtime given these counters.

There are several reasons why this is not straightforward. As seen in the previous section, there is not a simple linear relationship between number of flits and runtime, so linear regression is inappropriate. We only have counters for MILC's routers, but MILC's traffic is also sent through outside routers along non-deterministic paths. Collecting counters at the granularity of seconds loses information about the timing of the traffic. Finally, MILC occupies approximately 100 routers in a typical allocation, with 48 ports per router, so we have counter values for thousands of links which must be aggregated without losing critical information.

To help overcome these difficulties, we apply machine learning techniques to our data. In particular, we construct a set of simple statistical features from the flits and stalls counters

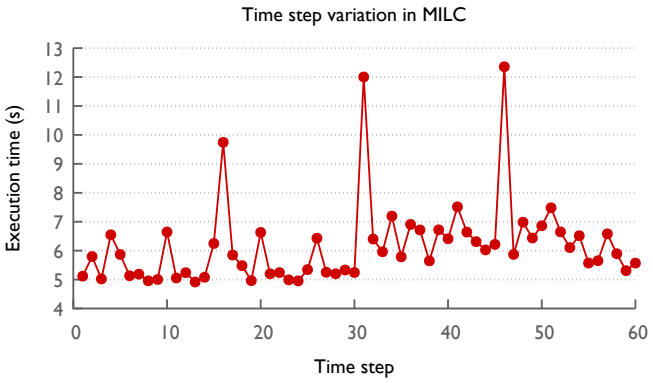


Fig. 10: Time per timestep for one execution of MILC.

and build a regression model with runtime 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 8 virtual channels across all nodes. We consider each timestep 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 timesteps in each run, respectively. Denoting $y_i$ as the runtime 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. This technique is appropriate for our data because it produces a non-linear model with an approximation that is often superior to other strategies, as explained below.

GBM is a machine learning paradigm where the key idea is to assume that the unknown function $\mathcal{F}$ is a linear combination of several *base learners*. The base learners will be greedily trained by setting their target response to be the negative gradient of the current loss with respect to the current prediction. Mathematically, we assume the function of interest,

$$y_i \approx \mathcal{F}(\mathbf{x}_i) = \sum_{j=1}^{m} \beta_j \psi_j(\mathbf{x}_i|\mathbf{z}_j), \tag{1}$$

where $\psi_j(\mathbf{x}|\mathbf{z}_j)$ is the base learner parameterized by $\mathbf{z}_j$. The GBM proceeds by performing a stage-wise greedy fit,

$$(\beta_j, \mathbf{z}_j) = arg\min_{\beta,\mathbf{z}} \sum_{i=1}^{n} L(y_i, \mathcal{F}_{j-1}(\mathbf{x}_i) + \beta\psi_i(\mathbf{x}_i|\mathbf{z})), \tag{2}$$

where $L$ is the loss function and $f_{j-1}$ is the estimate of the function obtained at the previous iteration,

$$\mathcal{F}_{j-1}(\mathbf{x}_i) = \sum_{t=1}^{j-1} \beta_t \psi_t(\mathbf{x}_i|\mathbf{z}_t). \tag{3}$$

The base learner coefficient $\beta_j$ is updated using steepest gradient descent. In our implementation, we consider the Huber loss function, and use simple decision trees as the base learner. Because GBM sequentially adds models by directly

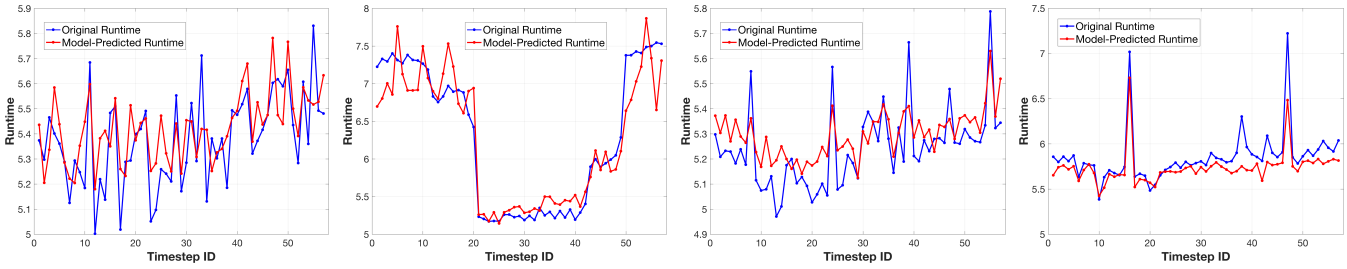


Fig. 11: Demonstration of the predictive power of the network counter data in describing the runtime behavior. Four executions and the predictions obtained from the network counters are shown here. The predictions were obtained using a GBM regressor with the sparse features.

optimizing for the negative gradient of the loss function, the final approximation is often superior to other strategies.

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. While PCA is ubiquitous, it is well-known that a few outlying data can grossly corrupt its results. A number of approaches to addressing this have been explored, and an important class of methods adopt a matrix decomposition approach 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. The following convex optimization problem is solved to estimate the low-rank and sparse components of $\mathbf{X}$:

$$\min_{\mathbf{L},\mathbf{S}} \|\mathbf{L}\|_* + \lambda\|\mathbf{S}\|_1 \text{ s.t. } \mathbf{X} = \mathbf{L} + \mathbf{S}, \tag{4}$$

where $\|.\|_*$ denotes the nuclear norm (convex surrogate for the rank of a matrix) and $\|.\|_1$ is the $\ell_1$ norm.

### B. Model Evaluation

We next evaluate our model to see if the network counters have predictive power in terms of runtime for the 87 runs we consider. First, we build regression models to predict the execution time for different timesteps in each run. To evaluate the performance, we perform 10-fold cross validation. The total set of runs is split into 10 partitions and in each fold nine of the partitions are used for training and the rest for testing. Figure 11 shows the actual runtimes and the model-predicted runtimes for several executions from the testing set. We can see that the model does a good job of predicting most timesteps in these executions. To quantify, we measure two different error metrics: (a) mean squared error (MSE), and (b) maximum absolute error (MAE) across all timesteps in each of the test runs. Table II shows the errors obtained using the base features (the matrix $\mathbf{X}$ directly) in comparison to the low-rank and sparse components from Equation 4. Furthermore, the regression performance obtained using a random forest regressor is reported for comparison. It can be observed that the GBM regressor outperforms a random forest regressor in both metrics. More interestingly, the sparse feature performs best because the outlying components in the counter values were the most useful in predicting the large discrepancies in the execution time.

| Feature | Gradient Boosted Machines | | Random Forest Regressor | |
|---|---|---|---|---|
| | MSE | MAE | MSE | MAE |
| Base | 0.56 (0.19) | 5.2 (3.23) | 0.75 (0.14) | 6.13 (2.74) |
| Low-Rank | 0.51 (0.22) | 6.02 (2.61) | 0.71 (0.09) | 6.22 (2.51) |
| Sparse | **0.35 (0.21)** | **5.04 (2.53)** | 0.53 (0.17) | 5.96 (2.62) |

TABLE II: Prediction performance - Mean Squared Error (MSE) and Maximum Absolute Error (MAE) of the predicted execution time obtained using GBM and random forest regressors. In each case, the average and standard deviation of the metrics obtained using a 10-fold cross validation are shown.

All versions of the model perform well on most runs, having a low mean squared error regardless of training/testing split used. From this, we have high confidence in the models' accuracy and conclude that the network counters have a relationship to the runtime variation we see in executions. Given our fine-grain trace findings, it is likely that the runtime variation in our experiments is caused by network traffic from other jobs, which is detected by the model via the counters.

However, despite having good average performance, the best model still has a relatively high max error on a handful of timesteps from our runs. Therefore, we explore whether we can make our predictions more accurate with a final modification.

### C. Improving Prediction with Local Models

Designing an accurate predictive model that generalizes well to variations in input features is challenging when the training data set is both limited and characterized by a large variance in the distribution of feature statistics. In such scenarios, it is beneficial to partition the data into multiple subsets and infer predictive models independently for each of the partitions.

While this approach leads to more accurate models, it is not straightforward to perform the partitioning and determine which of the models to apply to a new test sample. We address these challenges by (a) developing a novel subspace clustering algorithm to group similar runs, and (b) adopting the out-of-sample extension method for spectral clustering to our case.

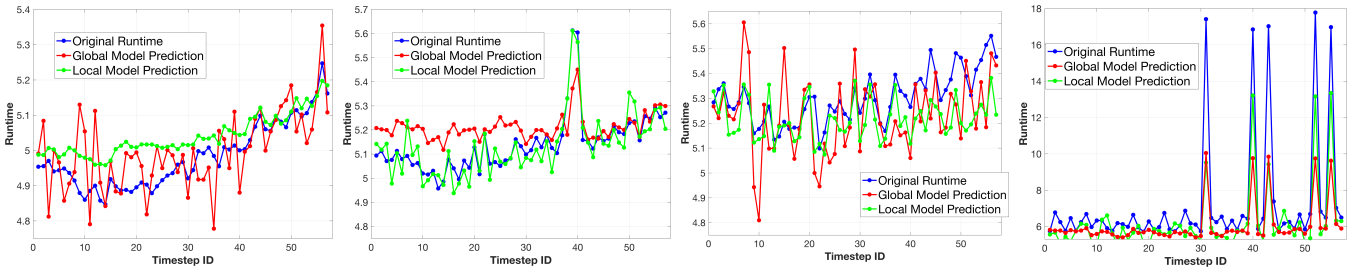


Fig. 12: Improved runtime prediction performance obtained by building multiple local regression models (2 in this case) to effectively handle runs with completely different network counter statistics. Note that different y-axis scales are used for readability.

Since each run corresponds to a matrix of feature values from all timesteps, we build a low-dimensional PCA subspace for the run and compare the subspaces for two different runs to measure the distance between them. We consider the two runs to be similar if (a) the two corresponding subspaces are geometrically well-aligned, and (b) the data in the projected subspaces are similarly distributed. More specifically, we use the following steps to measure distance between two runs:

1) Fit PCA subspaces of dimension $d$ (set to 5) to the two runs and compute the principal angles $\{\theta_i\}_{i=1}^{d}$ between them.
2) Project the input features onto the corresponding PCA subspaces.
3) For each dimension $i \leq d$: Fit two $1-$D Gaussians $\mathcal{S}_1^i$ and $\mathcal{S}_2^i$ to the $i^{\text{th}}$ dimension of the two subspaces, and compute the symmetricized KL-divergence between them as

$$\lambda_i = KL(\mathcal{S}_1^i||\mathcal{S}_2^i) + KL(\mathcal{S}_2^i||\mathcal{S}_1^i),$$

where

$$KL(\mathcal{S}_1^i||\mathcal{S}_2^i) = \sum_j \mathcal{S}_1^i(j) \log \frac{\mathcal{S}_1^i(j)}{\mathcal{S}_2^i(j)}.$$

4) Compute the distance measure as:

$$\text{dist}(\text{run}_1, \text{run}_2) = \frac{1}{d} \sum_i \theta_i \lambda_i.$$

The lower the distance, the more similar are the two runs. We use this novel distance measure to perform spectral clustering, which effectively partitions the runs into $K$ groups. For each group, we build a GBM regression model with the sparse features (recall that these features performed the best in Table II). We adopt the out-of-sample extension idea from Bengio et al. [14] to determine the appropriate group for an unseen test sample. Table III shows the MSE and MAE measures obtained using $K = 1, 2, 3$ regression models. We see that using even two local models results in a significant improvement in performance over the global predictive model, cutting the maximum prediction error nearly in half. We demonstrate the resulting predictions for four runs in Figure 12, and we clearly see the improvement in prediction performance obtained using local models.

| Number of Models | **1** | **2** | **3** |
|---|---|---|---|
| MSE | 0.35 (0.21) | 0.22 (0.09) | 0.2 (0.11) |
| MAE | 5.04 (2.53) | 2.87 (2.01) | 2.79 (1.86) |

TABLE III: Prediction performance obtained by training multiple regression models for different subsets of runs obtained by a novel clustering approach. Note that in all cases the sparse features were used to train GBM regressors.

## VIII. Related Work

The most closely related paper to ours studies job contention via simulation in dragonfly networks [5]. Their paper simulates execution on dragonfly machines, shows that contention exists, and then proposes a placement strategy to alleviate it. In contrast, we analyze interference and correlate it to specific network performance counters (flits and stalls) on a real dragonfly installation (Edison). Other researchers have used simulation to study dragonfly machines, including our own prior work [15] (which also provided a visualization tool to help understand dragonfly simulations).

Work in the same vein as ours includes the so-called "Neighborhood paper" [6]. In this paper, the authors showed that on a Cray torus machine, performance degradation occurred and was most likely due to nearby jobs. In contrast, our paper studies dragonfly networks and links slowdown to flits and stalls. Also work on "quiet neighborhoods" [16] proposed techniques to map jobs to virtual network blocks based on job size to try to avoid contention; this paper deals with real, current systems and their schedulers.

There are also simulators that deal with congested network links in supercomputers. For example, Hoefler et al. [17] developed LogGOPSim, which is a simulator that uses the LogGPS model in addition to contention. Other simulators that handle contention are SimGrid [18], TraceR [19] and SST/macro [20]. Overall, simulators are useful tools to understand aspects of contention, but real world systems tend to be more complicated than a chosen machine/system model.

## IX. Summary and Discussion

In this paper, we studied the performance variability of a real-world application on a modern supercomputer built with a dragonfly network interconnect. We found that the

application with which we experimented, MILC, experiences significant performance degradation when multiple jobs are competing for the network. This degradation occurs even though the dragonfly interconnect is supposed to help insulate applications from one another.

We analyzed the performance variability of MILC in relation to hardware network counters in two ways: (1) we performed a fine-grain analysis of an MPI trace of MILC to relate per-rank delays to heavily-subscribed links, and (2) we developed a coarse-grain model using machine learning to predict MILC's execution time given statistics of the network counters across the machine. Both analyses showed that slowdown in MILC is related to increased network activity. The model we have developed has a mean squared error of only 0.22 and could be used to help determine interference between various classes of applications and therefore help job schedulers schedule jobs with explicit regard for potential interference.

Despite its coarse granularity, our predictive model works quite well in most cases. That said, there are some limitations in the data available to us, and addressing these limitations would allow us to make a more precise analysis in the future. In particular, an ideal experimental environment would:

- allow us to collect network counters for all routers on the machine, rather than only ones local to our job (the LDMS [21] infrastructure exists for this purpose, but is not available on many machines);
- give us a better idea of the paths taken by our messages, so that we could analyze data for the exact links our application used and understand which links are being shared with other jobs;
- provide more precise network counters, such as counters for stalls which can be associated with the outgoing (stalling) port, rather than the incoming port on which the stalling message was received;
- give us the ability to sample counters at a finer granularity, to determine the precise timing of traffic;
- give us full control over the machine and the competing jobs.

Though communication interference between applications is an important yet poorly-understood problem, it is very difficult to study it in a real-world setting with the currently available tools. Tools that include some or all of the features above would help us perform an even deeper analysis.

## REFERENCES


[1] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the Scalability of Parallel Jobs by Adding Parallel Awareness to the Operating System," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03)*, 2003.

[2] D. Skinner and W. Kramer, "Understanding the Causes of Performance Variability in HPC Workloads," in *Proceedings of the IEEE International Workload Characterization Symposium, 2005*, 2005, pp. 137–149.

[3] 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.

[4] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[5] 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*, Nov. 2016.

[6] 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.

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

[8] 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.

[9] 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.

[10] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Supercomputing*, Nov. 2008.

[11] J. Moreira, M. Brutman, J. Castaños, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt, "Designing a highly-scalable operating system: The blue gene/l story," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[12] 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.

[13] "NERSC-8: Trinity Benchmarks." [Online]. Available: http://www.nersc.gov/users/computational-systems/nersc-8-system-cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks

[14] Y. Bengio, J.-F. Paiement, and P. Vincent, "Out-of-sample extensions for lle, isomap, mds, eigenmaps, and spectral clustering," in *In Advances in Neural Information Processing Systems*. MIT Press, 2003, pp. 177–184.

[15] A. Bhatele, N. Jain, Y. Livnat, V. Pascucci, and P.-T. Bremer, "Analyzing network health and congestion in dragonfly-based systems," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '16. IEEE Computer Society, May 2016, lLNL-CONF-678293.

[16] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: Key to protect job performance predictability," in *International Parallel and Distributed Processing Symposium*, May 2015.

[17] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, Jun. 2010, pp. 597–604.

[18] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.

[19] B. Acun, N. Jain, A. Bhatele, M. Mubarak, C. D. Carothers, and L. V. Kale, "Preliminary evaluation of a parallel trace replay tool for hpc network simulations," in *Proceedings of the 3rd Workshop on Parallel and Distributed Agent-Based Simulations*, ser. PADABS '15, Aug. 2015, lLNL-CONF-667225.

[20] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel architectures," *International Journal of Parallel and Distributed Systems*, vol. 1, no. 2, pp. 57–73, 2010.

[21] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *Supercomputing*, Nov. 2014.