The Case of Performance Variability on Dragonfly-based Systems

Abhinav Bhatele[†], Jayaraman J. Thiagarajan[*], Taylor Groves[‡], Rushil Anirudh[*], Staci A. Smith[§], Brandon Cook[‡], David K. Lowenthal[§]

[†]*Department of Computer Science, University of Maryland, College Park, Maryland 20742 USA*
[*]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA*
[‡]*NERSC, Lawrence Berkeley National Laboratory, Berkeley, California 94720 USA*
[§]*Department of Computer Science, The University of Arizona, Tucson, Arizona 85721 USA*
E-mail: [†]*bhatele@cs.umd.edu*

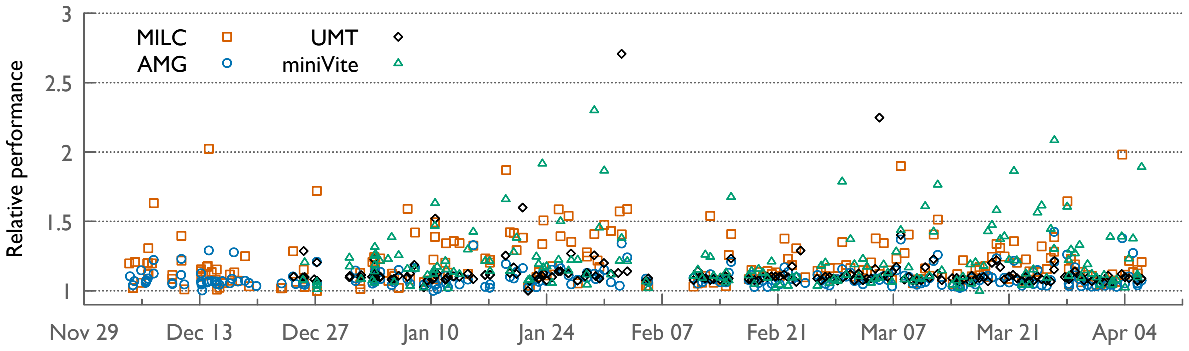

Figure 1. Variation in performance of four applications relative to their respective best observed run times when running on 128 nodes of Cori in 2018–2019.

***Abstract*—Performance of a parallel code running on a large supercomputer can vary significantly from one run to another even when the executable and its input parameters are left unchanged. Such variability can occur due to perturbation of the computation and/or communication in the code. In this paper, we investigate the case of performance variability arising due to network effects on supercomputers that use a dragonfly topology – specifically, Cray XC systems equipped with the Aries interconnect. We perform post-mortem analysis of network hardware counters, profiling output, job queue logs, and placement information, all gathered from periodic representative application runs. We investigate the causes of performance variability using deviation prediction and recursive feature elimination. Additionally, using time-stepped performance data of individual applications, we train machine learning models that can forecast the execution time of future time steps.**

***Keywords*-performance variability, dragonfly network, data analytics, machine learning, performance models, forecasting**

I. The Problem

Computational scientists who use large supercomputers for modeling science phenomena often submit many long running jobs over a period of time due to job time limits. These jobs often run the same executable and possibly the same input dataset. It is common to hear science users complain about performance variability within a single job from one time step to another and across equivalent jobs run over a period of time. Figure 1 shows that the performance of several HPC codes with different computational and communication characteristics can be up to 3× slower even when running the same executable and input. Such performance variability creates practical issues such as making performance debugging difficult and estimating the runtime of a job more challenging. More importantly, when jobs run slower than the best performance possible, they use resources for a longer amount of time than necessary, thereby reducing simulation efficiency and overall system throughput.

Performance variability in both short and long running jobs can arise from a multitude of factors ranging from operating system (OS) noise, varying network congestion, to filesystem (I/O) traffic. In this work, we primarily focus on variability arising from sub-optimal communication on networks that are shared by all concurrently running jobs. When network resources such as routers and links are shared by multiple jobs, it can lead to resource contention, which can degrade communication and I/O performance. This can significantly impact the overall performance of an individual job.

We target dragonfly-based systems in this paper because in spite of adaptive routing, such systems have been known to suffer from significant performance variability [1], [2]. Dragonfly systems are a popular network topology for deploying large supercomputers due to their low network diameter and high bandwidth. In order to study variability and identify its root causes, we set up controlled experiments using production and proxy applications running at different node counts. We perform our experiments on a Cray XC40 system at NERSC, Cori, which uses Aries routers to create a dragonfly topology.

In addition to application performance, we gather other related data such as time spent in computation and different

MPI routines, network hardware performance counters, job queue logs, and job placement information. We start with exploratory analysis to identify correlations between running jobs, their users, and their impact on the performance of our controlled experiments. We then use machine learning (ML) to find correlations between various independent features and the deviation in execution time from the mean behavior. The generated models are used to identify features that are strong predictors of execution time. Such analyses can be used by the resource manager to adapt scheduling decisions based on current system state.

Finally, we use ML to analyze time-stepped performance data of individual applications and to create models that can forecast the execution time of subsequent time steps. Using the generated models, we demonstrate that we can forecast the performance of both short controlled experiments and different time segments of long running production jobs. Such analysis can be extremely powerful in analyzing historical performance data being gathered at HPC facilities to forecast the system state in the future.

We make four main contributions in this paper:

- We perform controlled experiments on a dragonfly-based supercomputer to create a dataset of more than 1200 runs, each with multiple time steps.
- We identify correlations between performance slowdowns and concurrently running users and jobs on the system.
- We use ML to identify features that are important in predicting performance deviations within a run.
- We create a forecasting model that can predict future performance of a code based on its job placement, system state, and historical network counters data.

II. Background

Below, we describe the dragonfly topology, and potential sources of variability in HPC systems.

A. Dragonfly Topology and Cray XC systems

The dragonfly topology has become a popular choice for HPC interconnection networks because of low network diameter and high degree of connectivity, due to its use of high radix routers [3]. In this paper, we focus on Cray's implementation of the dragonfly topology in its Cascade (XC) line of supercomputers (e.g. Cray XC30 and XC40) [4]. Figure 2 illustrates the dragonfly topology of Cray XC systems, which use a 48-port router, called Aries, for connecting nodes in a two-level hierarchy. 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* or *row* links, and six routers in each column are also connected in an all-to-all configuration by *black* or *column* links. Each router is then connected using some *blue* or *global* links to routers in other groups.

The Cray XC systems use adaptive routing to evenly distribute network traffic over many links. When using adaptive routing, for any given packet, each router has multiple shortest and non-minimal paths to choose from. One of those paths

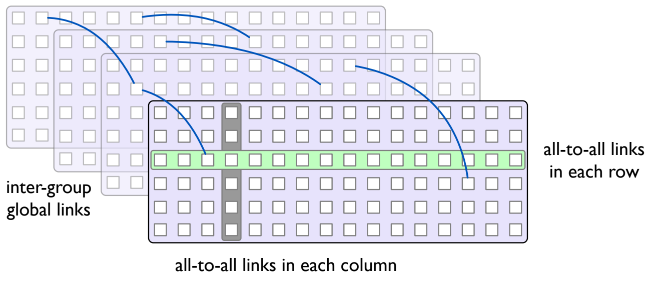

Figure 2. The dragonfly network configuration in Cray XC systems.

is selected based on the current back pressure observed on different links. We use Cori for all the experiments in this paper, which is a Cray XC40 installation at National Energy Research Scientific Computing Center (NERSC). Cori has 34 groups, seven of which have Intel Haswell nodes, and 27 have Intel Knights Landing (KNL) nodes. All of the experiments in this paper were run on the KNL nodes. Each KNL node contains a single-socket processor with 68 cores.

B. Sources of Performance Variability

With the increasing complexity of processing elements, networks, and software systems, variation within and between executions of the same program has become common. One oft-discussed form of variability is operating system (OS) noise [5]. Interruption of useful computation in a program by a system daemon can have negative effects downstream in programs that synchronize at a fine granularity [6]. This is referred to as OS noise. Several researchers have studied OS noise empirically and in many cases offered solutions (e.g., low-noise operating systems) to reduce or eliminate its effects.

Even in the absence of OS noise, there can be other contributors to performance variability on a supercomputer. For example, on most systems, the I/O subsystem is shared, and if two or more jobs access it simultaneously, degraded I/O performance can result. In this paper, we primarily focus on the effects of the interconnection network, which refers to application performance varying because of contention from other jobs using the shared network resources.

III. Data Collection

In this section, we describe the production and proxy applications used in the experiments on Cori, and the sources from which performance data is gathered for the study.

A. Application Codes and Inputs

We ran four codes that are representative of different HPC workloads commonly run at NSF and DOE centers:

AMG: The AMG proxy application is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. It is based on code in the *Hypre* linear solver library [7]. The input problem we ran in our experiments simulates a time-dependent loop with AMG-GMRES on a linear system built for a three-dimensional (3D) problem. The problem size per MPI process is 32 $\times$ 32 $\times$ 32 (see Table: I).

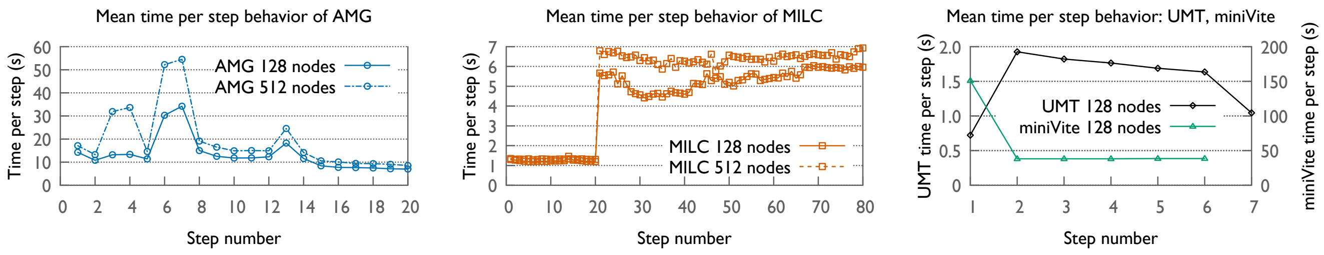

Figure 3. Plots showing the mean time per step behavior of each application across all runs.

TABLE I
APPLICATION VERSIONS AND THEIR INPUTS

Application	No. of Nodes	Input Parameters
AMG 1.1	128	-P 32 16 16 -n 32 32 32 -problem 2
AMG 1.1	512	-P 32 32 32 -n 32 32 32 -problem 2
MILC 7.8.0	128	n128_large.in
MILC 7.8.0	512	n512_large.in
miniVite 1.0	128	-f nlpkkt240.bin -t 1E-02 -i 6
UMT 2.0	128	custom_8k.cmg 4 2 4 4 4 0.04

MILC: MILC stands for MIMD Lattice Computation and is used to study quantum chromodynamics using numerical simulations [8]. We use the MILC application, `su3_rmd`, which performs the same amount of computation and communication in every time step (after a warmup phase). MILC performs a 4D Stencil on a per process grid of dimensions $4 \times 4 \times 4 \times 4$.

miniVite: miniVite is a proxy for the production application, Vite [9]. It performs a single phase of the Louvain classification for community detection in large distributed graphs. This is representative of new graph analytics workloads starting to run on HPC platforms. For this paper, we added another iterative loop in order to run the same computation and communication repeatedly. We use a real world graph called `nlpkkt240`, which has ~28 million vertices and ~373 million edges.

UMT: This is a discrete ordinates (S_n) code for multigroup deterministic, non-linear, radiation transport [10] over 3D unstructured spatial domains. The code allows user control over the number of energy groups and that of angles used in the discretization.

The experimental data for the paper was gathered by submitting one or two jobs (to the production job queue on Cori) per application and node count every day from a single user's account between December 2018 and April 2019. The actual start time for each submitted job was decided by the job scheduler, and sometimes, several of our jobs had an overlap in their execution periods. AMG and MILC were run on 128 and 512 nodes each because of their better scaling behavior, and miniVite and UMT were run on 128 nodes only. We used 64 out of 68 cores on each KNL node to set aside four cores for OS daemons. The inputs used for these runs are summarized in Table I. Each row in the table is considered an independent dataset with somewhere in between 175 and 225 runs in each.

B. Application Characterization

With the exception of miniVite, we ran each application for a certain number of iterations (time steps) to restrict the execution time to between five and ten minutes. As mentioned earlier, we added a second iterative loop in the case of miniVite to repeat the entire application execution six times. We recorded the time per step for each application. The first observable pattern, which happens to be true across all the applications, is that there is a mean time step behavior for each application and node count across all of its runs (Figure 3). Different runs of an application deviate from this mean behavior to different degrees, but the mean behavior is still discernible. In each execution, we also collected MPI profiles using mpiP to understand the time spent in computation and communication, and to identify the dominant MPI routines w.r.t. performance.

AMG: AMG runs for twenty time steps, and the mean time per step is shown in Figure 3 (left). AMG on 128 nodes performs better than on 512 nodes (we are using weak scaling) but the overall trends for time per step are similar. Figure 4 (left plots) shows the time spent in computation and communication on 512 nodes and the split of communication time into different MPI routines. The error bars on MPI time represent the slowest and fastest execution of AMG. We do not see a significant variation in compute time, which suggests the lack of OS noise. A significant amount of time is spent in MPI, which is due to the fact that we are stressing the scaling limits of AMG at 8,192 and 32,768 processes. The time spent in MPI varies significantly across runs, which causes overall performance variability. AMG sends a large number of small-sized messages and on average, spends 76 and 82% of the total time in MPI on 128 and 512 nodes respectively. We partition the MPI time into its constituent routines and observe that Iprobe, Test, Testall, Waitall, and Allreduce are the dominant routines.

MILC: The first twenty time steps in MILC are considered "warmup" trajectories and hence are much faster. The next sixty time steps are slower (Figure 3, middle). Note that even so, MILC time steps are shorter in duration than those of AMG and we are able to execute 80 of them in each run. MILC spends nearly 89% of its time in MPI on average (Figure 4, right plots). This is because we ran a relatively small input problem to limit the total execution time, which results in a significant fraction of the time spent in MPI. MILC sends large point-to-point messages, and the dominant MPI routines are

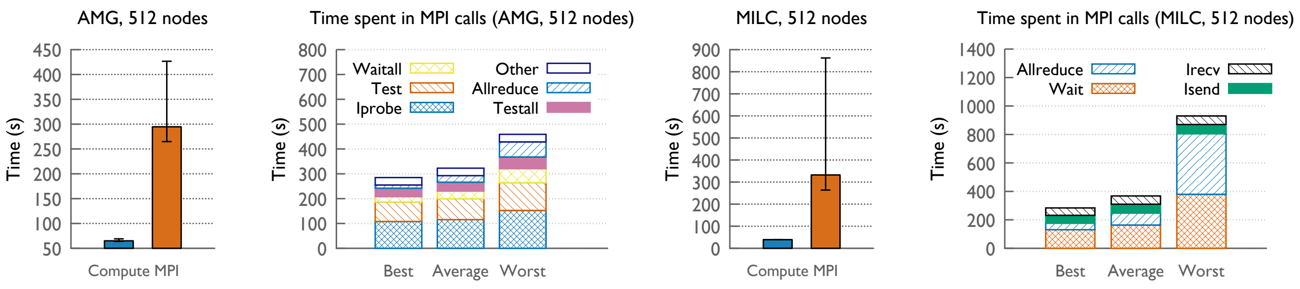

Figure 4. Time spent in computation and communication and in different MPI routines in AMG (left plots) and MILC (right plots) on 512 nodes.

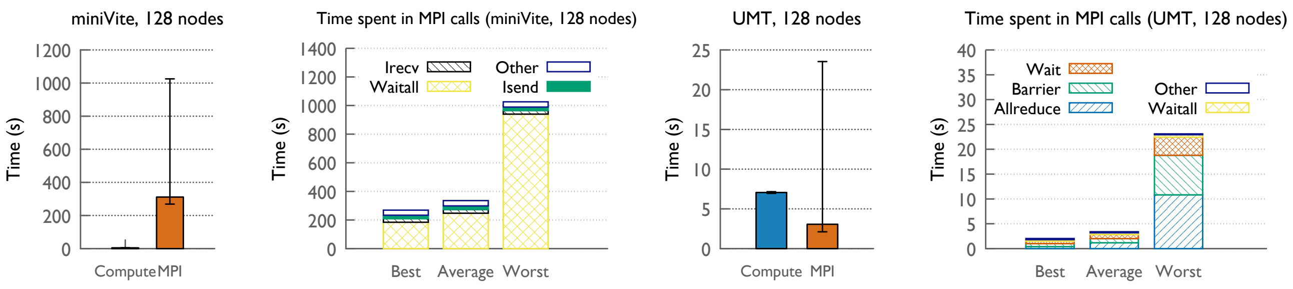

Figure 5. Time spent in computation and communication and in different MPI routines in miniVite (left plots) and UMT (right plots) on 128 nodes.

Allreduce, Wait, Isend and Irecv.

miniVite: miniVite spends more than 98% of its time in MPI on average (Figure 5, first plot). This is due to the inherent nature of the algorithm in miniVite, which requires frequent communication. Since most of the time is spent in MPI, variations in communication performance can be more drastic. The slowest recorded miniVite job ran 3.76× slower than the best performing job. Almost all of the MPI time in miniVite is spent in Waitall.

UMT: Among all the applications used in the paper, UMT has the smallest fraction of communication compared to its total execution time. However, it has some of the highest variability in performance. On an average, UMT spends 30% of its total time in MPI (Figure 5, third plot). However, the slowest recorded UMT job was 3.3× slower than the best performing UMT job. Most of the MPI time in UMT is spent in Allreduce, Barrier and Wait.

C. Sources of Performance Data

We gather hardware counters and related data from various sources to train the machine learning models for our analyses.

Network hardware performance counters: A wealth of performance counters are available on the Aries routers [11]. We use AriesNCL, a network counters library, which internally uses PAPI to collect counter data for our jobs. We record the execution time per step (iteration) of the application and the corresponding change in counter values. However, this has a limitation that users may only collect counters for routers that are directly connected to the nodes allocated to a job. Table II presents the network counters that are recorded and their descriptions. Processor tile counters (abbreviated names begin with `PT_`) are indicative of end-point traffic, i.e. data moving to and from NICs directly attached to a given router. Router tile counters (abbreviated names begin with `RT_`) capture data movement between network routers.

Much of the data we collect is focused on two things – (1) flits transmitted on the network, and (2) cycles stalled on various virtual channels (VCs). The stalls occur when the next hop in a route has no buffer space to receive additional flits or the arbiter is busy servicing other VCs. For system-wide data, i.e., counters on all routers in the system, we leverage LDMS, a monitoring service. LDMS data on Cori is collected every second, amounting to approximately 5 TB per day. We organize the system counters by the role of the nodes (compute versus I/O) attached to specific routers.

Job placement and neighborhood: In addition to network hardware counters, we derive several features from Slurm account logs (sacct). These job queue logs provide information about the nodes allocated to our jobs and jobs of other users in the system. We create two features for each of our jobs: `NUM_ROUTERS` is the number of unique routers to which the nodes in our job are attached, and `NUM_GROUPS` is the number of unique groups in the dragonfly topology on which a job's nodes are allocated. These features give an indication of the fragmentation of a job across routers and dragonfly groups. Using the job queue logs, we also track which other users and jobs were running during the same period as our jobs.

IV. Approach to Variability Analysis

We now describe the proposed analysis methodologies to identify factors responsible for the observed performance variability, and more importantly, to forecast future performance by leveraging statistics from the past data.

TABLE II
DESCRIPTION OF NETWORK HARDWARE PERFORMANCE COUNTERS (SOME ARE RAW AND OTHERS ARE DERIVED) USED IN THE STUDY.

Counter name	Abbreviation	Description
AR_RTR_INQ_PRF_INCOMING_FLIT_TOTAL	RT_FLIT_TOT	(Derived) Total number of flits received on router tile
AR_RTR_INQ_PRF_INCOMING_PKT_TOTAL	RT_PKT_TOT	(Derived) Total number of cycles stalled on router tile
AR_RTR_INQ_PRF_ROWBUS_2X_USAGE_CNT	RT_RB_2X_USG	Number of cycles in which two stalls occur on a router tile
AR_RTR_INQ_PRF_ROWBUS_STALL_CNT	RT_RB_STL	Total number of cycles stalled on router tile
AR_RTR_PT_COLBUF_PERF_STALL_RQ	PT_CB_STL_RQ	Number of cycles a processor tile is stalled for request VCs
AR_RTR_PT_COLBUF_PERF_STALL_RS	PT_CB_STL_RS	Number of cycles a processor tile is stalled for response VCs
AR_RTR_PT_INQ_PRF_INCOMING_FLIT_VC0	PT_FLIT_VC0	Number of flits received on processor tile on VC0
AR_RTR_PT_INQ_PRF_INCOMING_FLIT_VC4	PT_FLIT_VC4	Number of flits received on processor tile on VC4
AR_RTR_PT_INQ_PRF_INCOMING_FLIT_TOTAL	PT_FLIT_TOT	(Derived) Total number of flits received on processor tile
AR_RTR_PT_INQ_PRF_INCOMING_PKT_TOTAL	PT_PKT_TOT	(Derived) PT_RB_STL_RQ + PT_RB_STL_RS
AR_RTR_PT_INQ_PRF_REQ_ROWBUS_STALL_CNT	PT_RB_STL_RQ	Number of cycles stalled on processor tile request VCs
AR_RTR_PT_INQ_PRF_RSP_ROWBUS_STALL_CNT	PT_RB_STL_RS	Number of cycles stalled on processor tile response VCs
AR_RTR_PT_INQ_PRF_ROWBUS_2X_USAGE_CNT	PT_RB_2X_USG	Number of cycles in which two stalls occur on a processor tile

A. Neighborhood Analysis

First, we consider a coarse-grain analysis of performance variability of an application based on the knowledge of other application codes whose execution overlapped with that of the former ("the neighborhood"). Though this analysis does not take into account factors such as the placement of the concurrently executing jobs and the temporal extent of the actual overlap, we believe that meaningful correlations can exist between frequently occurring jobs and the *optimality* of an application. Though it might appear straightforward to apply a predictive modeling strategy (e.g., regression) to determine if the performance variability can be predicted solely based on the list of concurrently executing jobs or user IDs, such a model can be heavily biased because of spurious correlations. Hence, we propose to quantify the amount of information shared between each of the users (or jobs) and a given performance metric, and subsequently analyze only the top-ranked users (or jobs) to obtain a comprehensive view of their impact on the variability.

In our setup, we perform this coarse-grain analysis based on user IDs. We chose user IDs rather than job or executable names because those were not unique, and it was challenging to build an automated parser. We begin by creating a vocabulary $\mathcal{U}$ of all the users who have concurrently executing jobs with our application. For each independent run, r, of our application, we construct a binary vector $\mathbf{u}_r \in \mathbb{R}^{|\mathcal{U}|}, \forall r \in 1 \cdots N$, where $|\mathcal{U}|$ is the cardinality of the set of unique users, and N is the number of runs in one application dataset. The binary vector has a value of 1 for users with jobs executing concurrently with our application and zero elsewhere. Next, we define the *optimality* of each application run as follows: We measure the total execution time of a run, r as $t_r, \forall r \in 1 \cdots N$ and estimate its mean value t_m. For each run r, if $t_r < \tau t_m$, we mark that run as optimal ($\tau = 1$). Consequently, the performance metric (execution time) is transformed into a binary vector $\mathbf{p} \in \mathbb{R}^N$ denoting the optimality of the runs.

Given the user co-occurrence matrix $\mathbf{M}$ of size $N \times |\mathcal{U}|$, we quantify the dependency between each of the users and optimality based on mutual information (MI) [12]. Mutual information is a quantity that measures the relationship between two random variables that are sampled simultaneously. In particular, it measures how much information is communicated, on average, in one random variable about another. The formal definition of MI between two random variables X and Y, whose joint distribution is defined by $P(X, Y)$ is given as:

$$I(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(x,y) \log \frac{P(x,y)}{P(x)P(y)} \qquad (1)$$

Here, $\mathcal{X}$ and $\mathcal{Y}$ denote the sample set drawn simultaneously from the joint distribution, while $P(x)$ and $P(y)$ are the marginal densities. When the MI is equal to zero, the two variables are statistically independent, while higher values mean a higher degree of dependency. We obtain the mutual information between each column in $\mathbf{M}$ and $\mathbf{p}$, and the hypothesis is that users with the highest MI will be the most informative about performance optimality.

B. Modeling Performance Deviation

While the coarse-grain analysis provides insights into the impact of neighborhood on performance, a more fine-grain analysis is required to identify a subset of network hardware counters that best explain the observed performance variability. To this end, we treat each time step (corresponding to a single loop execution) from an application run as an independent sample and build a predictive model to estimate the time spent in that time step. We attempt to determine which counters are the most indicative of the efficiency of a particular time step. While existing approaches for such an analysis use machine learning techniques to build a mapping from the network counters to the time incurred, our focus is on describing the deviation from the *expected* behavior. In other words, instead of building a surrogate for application execution, we build a predictive model to estimate the variability. Interestingly, the factors responsible for variability are very different from those required for predicting the total execution time.

We denote the recorded network counters data for N independent runs of an application by the matrix $\mathbf{X}$ of size $N \times T \times H$, where T indicates the total number of time steps (i.e. number of times the loop is executed) and H is the number of network counters gathered (see Table II). The performance data or execution time for these runs is denoted by the matrix

$\mathbf{Y}$ of size $N \times T$. In order to obtain a model for the deviation from the mean behavior instead of the absolute times, we first remove the mean trends for both the counter data as well as the execution times. Consequently, we build the predictive models using the mean-centered data, $\hat{\mathbf{X}}$ and $\hat{\mathbf{Y}}$. Since we treat each time step as an independent sample, we transform $\hat{\mathbf{X}}$ and $\hat{\mathbf{Y}}$ into matrices of sizes $NT \times H$ and $NT \times 1$ respectively.

We use gradient boosted regression (GBR) [13] to build predictive models with cross-validation (10-fold), based on recursive feature elimination (RFE). We briefly review the formulation of GBR before describing the RFE technique. In gradient boosted machines, the key idea is to assume that the unknown regression function is a linear combination of several *base learners*. The base learners are greedily trained by setting their target response to be the negative gradient of the current loss with respect to the current prediction. The base learner can be imagined to be the "basis function" for the negative gradient. Concretely, let us assume the function of interest:

$$f(\mathbf{x}) = \sum_{j=1}^{\ell} \beta_j \psi_j(\mathbf{x}|\mathbf{z}_j) \tag{2}$$

where $\psi_j(\mathbf{x}|\mathbf{z}_j)$ is a base-learner (e.g. decision tree) parameterized by $\mathbf{z}_j$. The algorithm proceeds by performing a greedy fit:

$$(\beta_j, \mathbf{z}_j) = \arg\min_{\beta,\mathbf{z}} \sum_{i=1}^{N} L(y_i, f_{j-1}(\mathbf{x}_i) + \beta\psi_j(\mathbf{x}_i|\mathbf{z})) \tag{3}$$

Here L is the loss function and f_{j-1} is an estimate of the function obtained from the previous iteration.

With GBR trees as the underlying model, RFE is built upon the idea of repeatedly constructing a predictive model, identifying the worst performing feature (based on feature importance), setting that feature aside, and then repeating the process with the rest of the features. This process is applied until all features in the dataset are exhausted, and the features are ranked according to when they were eliminated. In effect, this is a greedy algorithm for finding the best performing subset of features. Finally, we compute the relevance score of each feature (network hardware counters in this case) as the likelihood of being chosen as a well-performing feature across all the cross-validation splits.

C. Forecasting Execution Time

The final critical component of our analysis approach is to build a forecaster, using multiple data sources, that predicts the performance of future time steps based on previous time steps. The underlying problem of time-series forecasting has long been studied in several fields from finance and climate to energy usage in power grids. Typically, this involves modeling the underlying dynamical process responsible for generating the time-series. Because estimating this dynamical process is not easy, a common approach for incorporating dynamic information is to include temporal context. This refers to information from previous time steps, in order to make predictions about the next time step. We adopt this formalism for forecasting performance of our applications. In the set of applications considered here, the performance (i.e. execution time) at each step can vary significantly. However, we expect longer term trends to be more reliably predictable.

We formulate the problem as one of predicting the aggregate performance of k future time steps, based on the network counters data from the last m time steps. The forecasting problem is formulated as follows: Denoting the set of input features (network counters) at any time step t by $\mathbf{x}(t)$, the forecasting problem is to predict $y^k_{tot}(t_c)$, the sum of execution times of the next k steps after the current time step t_c:

$$y^k_{tot}(t_c) = \mathcal{P}\left(\{\mathbf{x}(t)\}_{t=t_c-m}^{t_c}\right), \text{ where } y^k_{tot}(t_c) := \sum_{t=t_c+1}^{t_c+k} y(t)$$

Here $\mathcal{P}$ denotes the forecasting model. This formulation contains two free parameters: m, the number of historical steps that we consider for making predictions, and k, the number of steps that we are predicting. Figure 6 illustrates this formulation. For generating the training data, we use a sliding window based approach and "slide" t_c between m and $T - k$, where T is the maximum number of steps in each application run. Finally, we report the mean absolute percentage error (MAPE) in prediction across all our cross-validation splits. We find that such a forecasting model is highly generalizable, in that it can be used to predict across very long time frames even if we have used relatively shorter runs during training. The machine learning technique that we adopt in order to build $\mathcal{P}$ is the recently successful *attention models* in the deep learning literature [14]. Attention mechanism is a widely-adopted strategy in sequence modeling, wherein a parameterized function is used to determine relevant parts of the input to focus on, in order to make decisions about the future. We use the popular *scalar dot-product* attention along with a fully connected neural network to predict $y^k_{tot}(t_c)$.

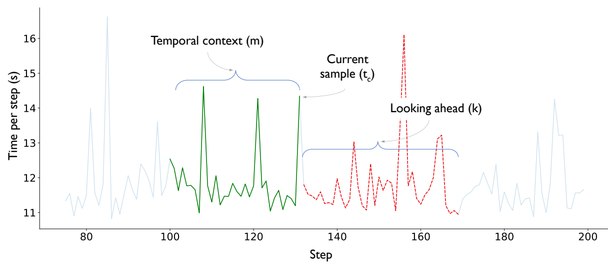

Figure 6. Illustration of the formulation for performance forecasting using different sources of application (job-specific), system and I/O data.

In order to understand how our forecasting capability varies, we conduct several ablation studies. First, we vary the temporal context used for prediction (m), and how far into the future we can predict (k). Then, we study how the prediction performance changes when we include information about the job placement (`NUM_GROUPS` and `NUM_ROUTERS`), and additional system-wide hardware counters (obtained from LDMS) for I/O nodes and "system" nodes (nodes allocated to jobs other than ours).

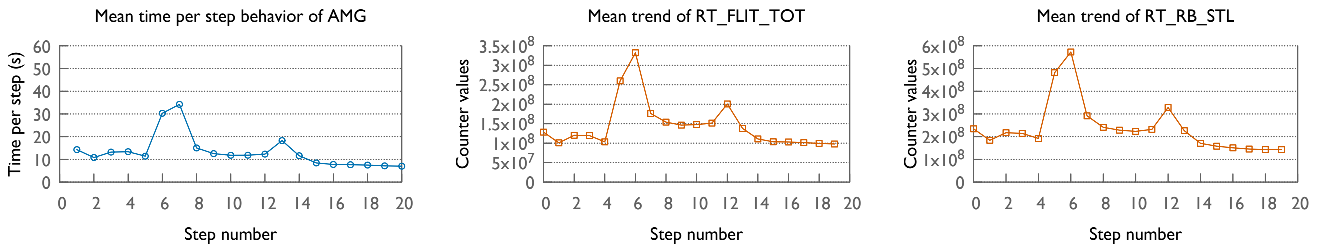

Figure 7. Mean time per step trends (left plot) are similar to mean counter values (middle and right plot) over all runs for each time step.

V. Findings

In this section, we apply the proposed machine learning methodologies to analyze the experimental data.

A. Neighborhood Analysis and Assigning Blame

In order to assign blame to users for performance degradation, for each job in an application dataset, we create a list of users that had one or more *running* jobs during the entire duration of our job (we define this as the "neighborhood" of a job). A user is considered only if their job size is larger than a certain number of nodes (128 for this analysis). These per-job lists give us an idea of the largest jobs running on the system alongside our job. We then quantify the dependency between optimality of jobs in each dataset and the "qualified" user names. Table III presents the lists of users that had high mutual information and appear in more than one list across the six datasets. Note that the actual user names have been anonymized.

TABLE III
Sets of highly correlated users (w.r.t. performance optimality) for the different datasets

Application	No. of nodes	Highly correlated users
AMG	128	User-[1, 2, 3, 4, 5, 6, 7, 8, 9]
AMG	512	User-[2, 3, 7, 9, 10, 11, 12]
MILC	128	User-[2, 8, 9, 11, 13, 14]
MILC	512	User-[8, 10, 11, 14]
miniVite	128	User-[2, 4, 5, 8, 12, 13]
UMT	128	User-[1, 6, 11]

We observe that some users appear in multiple rows of the table above. Users 2, 8 and 11 appear in four lists as being highly negatively correlated with optimality. This suggests that if one of these users is running a job on the system, there is a good chance that other communication-heavy jobs will slow down. User 9 shows up in three lists, and the other users show up in two lists each. User 8 is Bhatele, who submitted all the jobs for this study; in other words, two different jobs that we submitted can interfere with each other. For example, we know that MILC is communication-heavy and can create congestion on the network.

It so happens that **all** jobs of each of the users in Table III had the same or very similar job names, and we were able to identify the applications being run by these users in several instances. User 2 ran HipMer, which is a Genome Assembly code that is both communication-intensive and performs heavy I/O to the filesystem. User 11's jobs were doing climate modeling using the Energy Exascale Earth System Model (E3SM) code. User 9 was running a particle mesh N-body solver, FastPM, which invokes `MPI_All_Reduce` many times and also uses burst buffers for I/O. Many other users (6, 10 and 14) ran material science simulations. These applications are also known to send significant MPI and/or filesystem traffic on the network. Hence, there is great likelihood that these jobs will impact the performance of other jobs.

This suggests that even a coarse-grain analysis of total execution time and running jobs on the system can help us in identifying potential users/jobs that can impact performance negatively. A resource manager can use such historical data to delay scheduling jobs that are communication-sensitive when certain other jobs are already running on the system.

B. Identifying Strong Predictors of Deviation

Next, we attempt to identify network counters that are strong predictors of performance variability. As described in Section III-C, for each job, we record the execution times and corresponding counter value changes per step (iteration). In this analysis, we consider each time step as an independent sample within each of the six datasets. Note that we consider the deviation from mean behavior instead of absolute times to identify network counters that are strong predictors of run-to-run variability. We did this because we observed that the trends of the mean values of many counters and mean execution time per step are similar across all the runs (Figure 7).

Using recursive feature elimination and gradient boosted regression for the prediction model, we compute the relevance scores of different features (counters) independently for each of the six datasets (Figure 9). We observe that the stall counter on router tiles (`RT_RB_STL`) is highly relevant for the MILC datasets and the AMG 512 nodes dataset. The stall counter gives an aggregate view into the backpressure incurred by the network traffic received on the router tiles. In addition, the processor tile stall counter, `PT_CB_STL_RS`, has moderate relevance for MILC, and `PT_RB_STL_RQ` and `PT_RB_2X_USG` are relevant for AMG. This suggests that to a smaller extent for MILC, and a larger extent for AMG, end-point congestion in the processor tiles impacts performance. This becomes even more pronounced for UMT, where the `PT_RB_STL_RQ` counter is the most significant. Due to the nature of communication in miniVite, flit counters at the processor and router tiles (`PT_FLIT_VC0`, `RT_FLIT_TOT`) are most important for it.

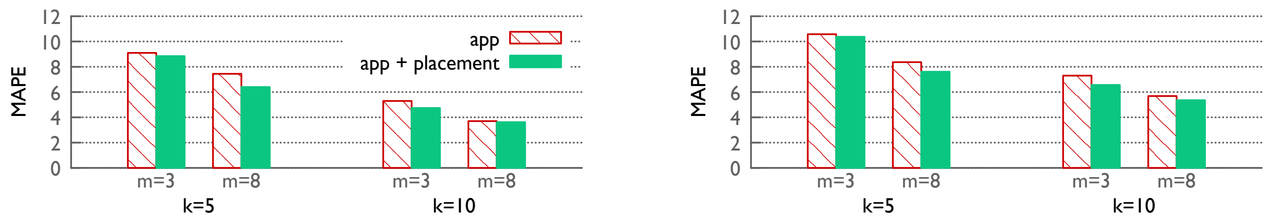

Figure 8. Mean absolute percentage error of the forecasting model for different m and k for the AMG 128 node (left) and 512 node (right) datasets.

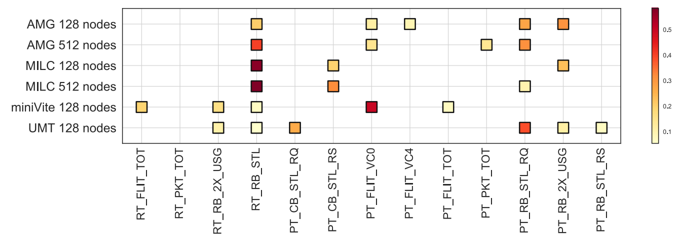

Figure 9. Relevance scores of each counter in predicting the deviation from mean behavior for the different datasets.

The mean absolute percentage error (MAPE) made by the prediction models was less than 5% for all the datasets. This analysis helps us identify, for different applications, which network hardware performance counters are most relevant in predicting deviation from the mean behavior. This motivates the possibility of using these features to forecast future performance based on past behavior. Additionally, because monitoring and analyzing hundreds of counters in real time has high overheads, the results of such analysis can help identify the critical counters that should be used by job schedulers in deciding when and where to schedule pending jobs.

C. Forecasting Execution Time

We now analyze the accuracy of the forecaster developed in Section IV-C in predicting the execution time of k future time steps using counter data from m previous time steps. So far we only considered network counters for routers directly attached to our jobs' nodes as input features. We now consider additional data to include features that contain information about the job placement and other routers on the system:

- **placement**: These features (`NUM_ROUTERS` and `NUM_GROUPS`) depend on the placement of the job and indicate the degree of fragmentation of the job.
- **io**: LDMS collects four counters in Table II. The "io" features refer to data collected by LDMS from I/O nodes on Cori that connect to the filesystem. These counters give an indication of the filesystem traffic on the network.
- **sys**: These are also derived from LDMS data gathered from all routers on the system that have no nodes in common with those allocated to our job. Values of these counters give an indication of the traffic on other routers of the system.

We create multiple independent models for different values of m and k, where m is kept small relative to the total number of time steps so that we have enough samples in the training set. The value of k is set to 25% and 50% of the total number of time steps in order to assess the ability to predict up to half of the execution period of a job. Based on these considerations, we choose $m = \{3, 8\}$, $k = \{5, 10\}$ for AMG, and $m = \{10, 20\}$ and $k = \{20, 40\}$ for MILC. Note that the absolute execution time of time steps in AMG is much greater than those in MILC. So, even though we use larger values of m in the case of MILC, the temporal context used for modeling in terms of the absolute time is similar to that of AMG. We did not perform forecasting for miniVite and UMT because they ran for six and seven time steps respectively, which was not long enough to create reasonable models.

Figure 8 shows the MAPE for the different forecasting models created for the two AMG datasets. We observe that a longer temporal context (larger m) lowers the MAPE significantly. In addition, larger values of k allow bursty performance changes per time step to be amortized, and so predictions improve as k increases. We see similar trends for the 128 and 512 node datasets. The 512 node datasets have slightly higher errors, possibly because of the larger variation in performance at that scale. We do not use the io and sys features for AMG because they lead to overfitting. We do not see a significant improvement in forecasting by adding placement features.

Figure 10 shows the MAPE for the different forecasting models created for the two MILC datasets. In this case, we use the placement as well as io and sys features. The observations from the AMG forecasting models for different values of m and k still hold true. The largest difference from the AMG forecasting models is that for MILC, adding io and sys features successively lowers the errors and makes the forecasting much better. We believe that this is because MILC is bandwidth-bound, and increased I/O and MPI traffic on the system impacts its performance more than AMG. This suggests that in addition to a job's own routers, traffic on other routers of the system can also impact its performance.

Next, we look at feature importances derived from the forecasting models. Note that as opposed to the previous section where the goal was to predict the deviations in performance, the forecasting models are trying to predict the absolute performance. Also, in the case of MILC, we are now using a much larger set of input features. Figure 11 shows the feature importances in the case of AMG and MILC for the largest m and k considered in each case. We observe that in the case of AMG (left plot), `PT_RB_STL_RQ` is no longer relevant and `PT_RB_STL_RS` now has high relevance for AMG 512 nodes.

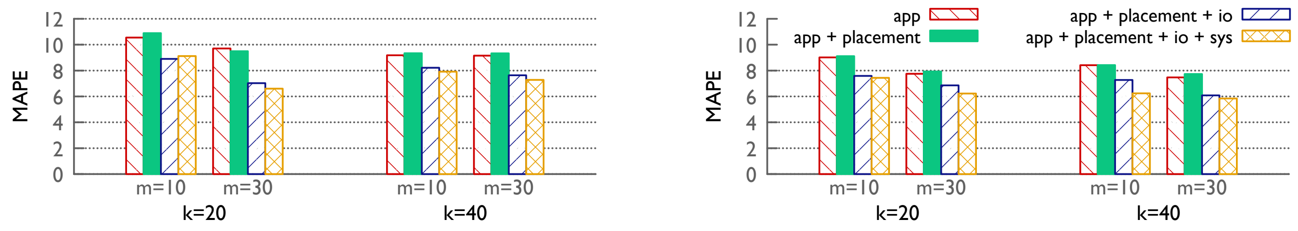

Figure 10. Mean absolute percentage error of the forecasting model for different m and k for the MILC 128 node (left) and 512 node (right) datasets.

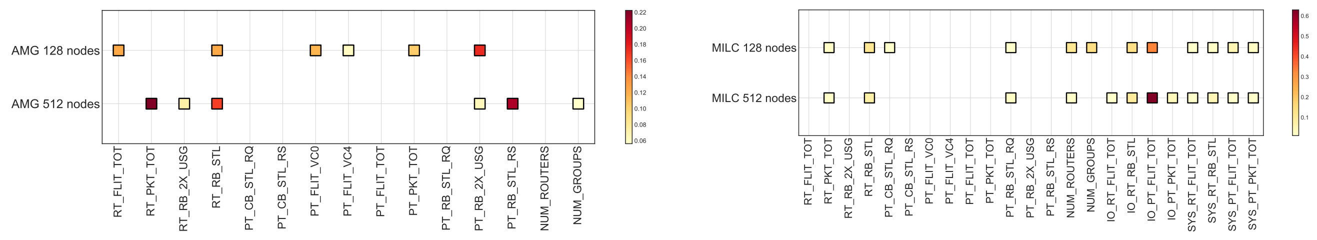

Figure 11. Feature importances derived from the forecasting model for the AMG (left) and MILC (right) datasets.

RT_RB_STL is still important but now, flit-based features (RT_FLIT_TOT and RT_PKT_TOT) are also relevant for the 128 and 512 datasets, respectively.

In the case of the MILC datasets, we previously observed that the job-specific router tile stalls counter RT_RB_STL had the highest relevance in terms of predicting performance deviation. Interestingly, in the forecasting model, we observe that the io router tile flit counter (IO_PT_FLIT_TOT) has the highest relevance for MILC, and all other features become less relevant. This counter gives an indication of how much I/O traffic is being sent to the I/O nodes from all over the system, and we believe that for MILC, this is a strong predictor of future performance.

Finally, we analyze the effectiveness of the generated forecasting models in predicting the execution times of future time steps in a significantly long-running simulation. To test this, we ran a MILC job on 128 nodes for 620 time steps, which took more than an hour and 45 minutes. We divide this execution into segments of 40 time steps and predict the execution time of these segments by considering only the previous 30 time steps. It is important to note that no data from this run was included in training the model. Figure 12 shows that using the generated models, we are able to forecast the performance of a long running science job in the face of significant performance variability. Note that, given the inherent uncertainties in this forecasting problem, there is irreducible bias in the predictive model, which manifests as larger prediction error for some time periods.

We believe that this is a significant finding in this work – we are able to use historical information about an application to predict the time-stepped performance of unseen data in a long running job. This indicates that time-series forecasting could be used to train models on historical performance data from an HPC system. Such models can then be used by system administrators or resource managers to forecast future system state such as MPI traffic or I/O load on the system, which

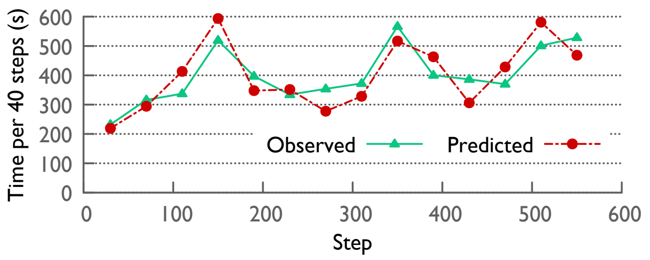

Figure 12. Forecasting 40 step segments using $m = 30$

can be used in turn to influence scheduling and other resource management decisions for system optimization. For this to be successful, the resource manager would need application and system data from the past to train the models. When using these models, the resource manager would need information about the current state of all the routers and additional information about the jobs in the queue (to ensure that the executable, input parameters, etc. match the setup used for training).

VI. Related Work

Several researchers have studied performance variation on systems using dragonfly interconnects. Chunduri et al. [2] perform a run-to-run variability study on a Cray XC system. However, they primarily focus on on-node variability (which itself is well studied [5], [6]), whereas we focus on network variability. Groves et al. [15] examine the correlation of user-level counters to simple network benchmarks on the Aries network, using simple linear regression. However, neither of these works attempt prediction of application variability from counter measurements. Tuncer et al. [16] perform classification and detection of anomalous performance based on Aries counters. The system evaluated is significantly smaller (52 nodes) and synthetic congestion patterns are used.

Other researchers have studied network variation via simulation and modeling. For example, Yang et al. [17] simulate execution on dragonfly machines, show that contention exists, and then propose a placement strategy to alleviate it. Faizian et al. [18] present simulated results of using software defined routing on a dragonfly and compare to adaptive routing. There is also work on studying and alleviating interference on other interconnects; these are complementary to our work. For example, Bhatele et al. [19] show that on a Cray torus machine, performance degradation occurs and is most likely due to nearby jobs. Smith et al. [1] and de Sensi et al. [20] propose modifications to the routing policy to alleviate congestion on fat-tree and dragonfly networks respectively. Jha et al. [21] investigate the relationship between counters and observed performance on the Gemini network.

VII. Conclusion

In this work, we addressed the difficult challenge of analyzing and predicting performance variability on a production system with a dragonfly network. The evaluated system supports thousands of unique users and workloads, and our analysis encompassed over four months of application runs. Despite these challenges, our approach was fruitful, and this work makes a number of contributions to the area. We have developed a methodology to identify the users and jobs correlated with runtime variability of our workloads. We are able to identify the key network hardware counters responsible for variability due to network congestion for the applications examined, as evidenced by our deviation prediction models. Finally, our forecasting models provide accurate predictions spanning up to 50% of the application runtime. We identify the key metrics that impact application runtime and observe that most of the predictive power comes from counters of routers that are directly connected to the nodes of a job. We also observe I/O traffic having a significant impact on network performance of bandwidth-bound codes. In future work, we plan to exploit this predictive power to improve scheduling and placement.

Acknowledgment

This work was supported by funding provided by the University of Maryland College Park Foundation, and based upon work supported by the National Science Foundation under Grant No. 1526015. This work was performed in part under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-772401), and used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] S. Smith, C. Cromey, D. K. Lowenthal, J. Domke, N. Jain, and A. Bhatele, "Mitigating inter-job interference using adaptive flow-aware routing," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. IEEE Computer Society, Nov. 2018, lLNL-CONF-745538.

[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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017.

[3] 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.

[4] 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.

[5] 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.

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

[7] R. Falgout, J. Jones, and U. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. Bruaset and A. Tveito, Eds. Springer-Verlag, 2006, vol. 51, pp. 267–294.

[8] 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.

[9] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 885–895.

[10] P. F. Nowak and M. K. Nemanic, "Radiation transport calculations on unstructured grids using a spatially decomposed and threaded algorithm," in *Proceedings of the International Conference on Mathematics and Computation, Reactor Physics and Environmental Analysis in Nuclear Applications*, Sep. 1999.

[11] "Aries hardware counters (s-0045-20)," http://docs.cray.com/books/S-0045-20/S-0045-20.pdf, 2017.

[12] S. Kullback, *Information theory and statistics*. Courier Corporation, 1997.

[13] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, no. 5, pp. pp. 1189–1232, 2001.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[15] T. Groves, Y. Gu, and N. J. Wright, "Understanding performance variability on the aries dragonfly network," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 809–813.

[16] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Online diagnosis of performance variation in hpc systems using machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 883–896, April 2019.

[17] 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.

[18] 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.

[19] 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.

[20] D. D. Sensi, S. D. Girolamo, and T. Hoefler, "Mitigating Network Noise on Dragonfly Networks through Application-Aware Routing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019.

[21] S. Jha, J. Brandt, A. Gentile, Z. Kalbarczyk, and R. Iyer, "Characterizing supercomputer traffic networks through link-level analysis," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 562–570.