LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-663150

Identifying the Culprits behind Network Congestion

A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P. T. Bremer, M. Schulz, L. V. Kale

October 22, 2014

29th IEEE International Parallel & Distributed Processing Symposium
Hyderabad, India
May 25, 2015 through May 29, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Identifying the Culprits behind Network Congestion

Abhinav Bhatele[†], Andrew R. Titus[*], Jayaraman J. Thiagarajan[†], Nikhil Jain[§], Todd Gamblin[†],
Peer-Timo Bremer[†,‡], Martin Schulz[†], Laxmikant V. Kale[§]

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA
[*]Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA
[§]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL
[‡]Center of Extreme Data Management, Analysis and Visualization, University of Utah, Salt Lake City, UT
E-mail: [†]{bhatele, jayaramanthi1, tgamblin, ptbremer, schulzm}@llnl.gov, [*]atitus@mit.edu, [§]{nikhil, kale}@illinois.edu

***Abstract*—Network congestion is one of the primary causes of performance degradation, performance variability and poor scaling in communication-heavy parallel applications. However, the causes and mechanisms of network congestion on modern interconnection networks are not well understood. We need new approaches to analyze, model and predict this critical behavior in order to improve the performance of large-scale parallel applications. This paper applies supervised learning algorithms, such as forests of extremely randomized trees and gradient boosted regression trees, to perform regression analysis on communication data and application execution time. Using data derived from multiple executions, we create models to predict the execution time of communication-heavy parallel applications. This analysis also identifies the features and associated hardware components that have the most impact on network congestion and in turn, on execution time. The ideas presented in this paper have wide applicability: predicting the execution time on a different number of nodes, or different input datasets, or even for an unknown code, identifying the best configuration parameters for an application, and finding the root causes of network congestion on different architectures.**

***Keywords*-interconnection network, congestion, root cause, machine learning, modeling, performance prediction**

I. Motivation and impact

Network congestion is widely recognized as one of the primary causes of performance degradation, performance variability, and poor scaling in communication-heavy applications running on supercomputers [1], [2], [3], [4], [5]. However, due to the complex nature of interconnection networks, as well as message injection and routing strategies, network congestion and its root causes in terms of network resources and hardware components are not well understood. This makes the problem of mitigating and avoiding network congestion difficult. It also complicates the task of writing congestion-avoiding and congestion-minimizing algorithms for communication and task mapping. Therefore, we need new approaches to understand and model network congestion in order to improve the performance of large-scale parallel applications.

When a message is sent from one node to another, it is split into packets that pass through many resources and hardware components on the network. A packet starts in an injection FIFO on the source. It then passes through multiple network links and receive buffers on intermediate nodes before it finally lands in the reception FIFO on the destination. When shared by multiple packets, any or all of these network components can slow down individual flits, packets and messages. This paper aims to identify the hardware components that affect the performance of a message send the most.

Our approach is based on using supervised machine learning to build models that map from independent variables, representing different network hardware components, to a dependent variable – the execution time of the application. We only consider computationally balanced, communication-heavy parallel applications and, hence, focus on the communication fraction of the total execution time. In order to generate multiple different executions of a parallel application, we vary the placement or layout of application processes/ tasks on the network. The different *task mappings* result in different message flows on the network and different execution times. This allows us to measure network hardware counters and execution times for the same executable under different configurations and network conditions.

Using supervised learning algorithms such as forests of extremely randomized trees [6] and gradient boosted regression trees [7], we perform regression analysis on communication data, derived from network hardware counters and analytical modeling, and the corresponding execution time. In addition to predicting the execution time for new samples, this analysis also identifies the features and associated hardware components that have the most impact on network congestion and in turn, on the execution time.

Sukhija et al. use supervised learning algorithms to select the best dynamic loop scheduling algorithm for parallel loops to improve on-node performance [8]. Tallent et al. use measurement and modeling to diagnose the cause of end-point contention for one-sided messages [9]. In our previous work [10], we presented a preliminary study on predicting the execution time and relative performance of different task mappings using forests of randomized trees [11]. This paper builds on our previous work and makes the following contributions:

- We exhaustively search all possible combinations of features using several supervised learning techniques to identify those that lead to the best predictions.
- Using different thresholds for a quantile loss function, we show that different features are important in different performance regimes, i.e., in fast vs. slow executions.

- We analyze the relative impact of different features on predicting execution times to identify hardware components that contribute the most to network congestion.
- We demonstrate our technique using various communication kernels as well as two scalable, communication-heavy applications, MILC [12] and pF3D [13].

The prediction techniques presented in this paper are widely applicable to a variety of scenarios, such as, (1) creating offline prediction models that can be used for low overhead tuning decisions to find the best configuration parameters, (2) predicting the execution time in new setups, e.g., on a different number of nodes, or different input datasets, or even for an unknown code, (3) identifying the root causes of network congestion on different architectures, and (4) generating task mappings for good performance.

II. Potential root causes of network congestion

When messages travel over the interconnection network, they are broken into smaller units: packets, chunks and flits. These pass through various hardware components, any or all of which can delay the communication [10]. We briefly explain the hardware components and the measurements that we would need to evaluate contention on each of them (see Table I).

TABLE I
HARDWARE COMPONENTS POTENTIALLY RELATED TO NETWORK CONGESTION AND THEIR CORRESPONDING INDICATORS

Hardware resource	Contention indicator
Source node	Injection FIFO length
Network link	Number of sent packets
Intermediate router	Receive buffer length
All	Number of hops (dilation)

At the source node, a message is split into several packets that are enqueued in network injection FIFOs (there are several FIFOs per node). Depending on the algorithm used to assign packets to injection FIFOs, there may be contention for these FIFOs among packets of one or multiple messages. From the injection FIFOs, packets are transferred to network links, which are typically shared by many messages when multiple routes pass through the same links. When multiple messages share the same links, the effective bandwidth is less than the peak due to link contention.

Packets may stall on a router because the next link is busy or because the destination node is unable to process incoming packets at their arrival rate. When this happens, routers store packets temporarily in receive buffers. Stalled packets may cause congestion when these buffers become full. Finally, each intermediate component that a message passes through along its route increases the chance of network congestion. So, the number of hops a message travels, also referred to as *dilation*, can also be an important indicator of congestion.

III. Methodology and experimental setup

In this section, we describe the process of gathering and preparing the input data for machine learning, the communication kernels and applications we use, and the step-by-step methodology we have developed to apply supervised learning algorithms to train our models.

A. Gathering data for supervised learning

The goal of this paper is to find correlations of network and communication related metrics with application execution time. In machine learning terms, the metrics are *features*, or inputs to machine learning algorithms, and the execution time is the *dependent variable*. Our dataset thus consists of a tuple of features and the execution time for each experiment, which is called a *sample* in machine learning. Each sample or experiment is a single run of the application.

We use the technique of task mapping to create a dataset of several samples for each application that is large enough to be statistically meaningful. Task mapping allows us to change the placement of application processes on the network, thereby changing the flow of messages and the corresponding execution time. This allows us to collect network hardware counters and execution time for different configurations of running the same application executable.

In this paper, we focus on torus interconnects, in particular, on the five-dimensional (5D) torus network, which provides an interesting experimental testbed to study the effects of network congestion. All the experimental data for this study has been collected on Vulcan, an IBM Blue Gene/Q installation at LLNL. We use Rubik [14] to generate many different task mappings of the code running on a 5D torus.

TABLE II
LIST OF COMMUNICATION METRICS (FEATURES) USED AS INPUTS TO THE MACHINE LEARNING MODEL. THE COLORS IN THIS TABLE CORRESPOND TO DIFFERENT HARDWARE COMPONENTS IN TABLE I

Feature name	Description
avg dilation AO	Avg. dilation of average outliers (AO)
max dilation	Maximum dilation
sum dilation AO	Sum of dilation of AO
avg bytes	Avg. bytes per link
avg bytes AO	Avg. bytes per link for AO
avg bytes TO	Avg. bytes per link for top outliers (TO)
max bytes	Maximum bytes on a link
#links AO bytes	No. of AO links w.r.t. bytes
avg stalls	Avg. receive buffer length
avg stalls AO	Avg. receive buffer length for AO
avg stalls TO	Avg. receive buffer length for TO
max stalls	Maximum receive buffer length
#links AO stalls	No. of AO links w.r.t. recv buffer length
avg stallspp	Avg. number of stalls per rcv'd packet
avg stallspp AO	Avg. no. of stalls per packet for AO
avg stallspp TO	Avg. no. of stalls per packet for TO
max stallspp	Maximum number of stalls per packet
#links AO stallspp	No. of AO links w.r.t. stalls per packet
max inj FIFO	Maximum injection FIFO length

Based on the list of hardware components that could contribute to network congestion (Table I), we gather communication data from three network hardware counters: the number of packets sent on each link, the receive buffer length and the

TABLE III
SIZES OF THE INPUT DATASETS IN TERMS OF THE NUMBER OF EXECUTIONS OR SAMPLES FOR THE DIFFERENT CODES

	2D Halo		3D Halo		Sub A2A		MILC	pF3D	Total
#Nodes	16 KB	4 MB	16 KB	4 MB	16 KB	4 MB			
1024	84	84	84	84	84	84	208	94	806
4096	84	84	84	84	84	84	103	103	710
Total	168	168	168	168	168	168	311	197	1516

number of packets received on each link. We use analytical modeling to obtain data for two other sources: the injection FIFO length and the dilation for each message. Data from these five sources is broken down into nineteen features and grouped into five categories denoting the source: dilation, bytes, stalls, stallspp, injFIFO (see Table II). The receive buffer length is referred to as *stalls* in the rest of the paper because it indicates the number of times different packets are stalled on intermediate nodes. *Stallspp* refers to the average number of stalls observed per packet.

The raw data we obtain for each execution is gathered *per link* in the network. To train our models, we require a single value for each feature aggregated over all the links. To achieve this, we use aggregates such as the average or maximum value of a feature over all links. We also consider a smaller subset of links from the distribution, such as only those with a value greater than the mean (average outliers or AO), or those that are in the top 5% of the distribution (top outliers or TO). This helps us create several different aggregated features for each source or hardware component from which we obtained raw data. In the end, each execution (one sample) is represented by the nineteen features shown in Table II and a corresponding execution time.

B. Description of parallel codes used

We use three different communication kernels and two scalable, communication-heavy, production applications for the analysis in this paper. A brief introduction to each is provided below:

Five-point 2D halo exchange: The *2D Halo* communication kernel uses a 2D grid of MPI processes to exchange four messages with two neighbors in each dimension.

15-point 3D halo exchange: The *3D Halo* communication kernel uses a 3D grid of MPI processes to exchange fourteen messages with its near-neighbors (six faces and eight corners).

All-to-all over sub-communicators: The *Sub A2A* communication kernel also uses a 3D process grid but performs all-to-alls on sub-communicators of size 64, formed from processes in one of the three dimensions.

MILC: MILC [12] is a Lattice Quantum Chromodynamics (QCD) application that does near-neighbor exchanges over a 4D process grid, similar to 2D and 3D Halo.

pF3D: pF3D [13] is a laser-plasma interaction code that performs all-to-alls over sub-communicators (similar to Sub A2A) and near-neighbor exchanges over a 3D process grid.

The communication kernels are executed with two different message sizes – 16 KB and 4 MB to evaluate different MPI performance regimes. The computational load of both MILC and pF3D is almost perfectly balanced across MPI processes. This allows us to focus on their communication, which is a significant portion of their overall execution time. We ran all the codes on 1024 and 4096 nodes of Blue Gene/Q to study the congestion behavior on different torus sizes. Depending on the code, we placed between 16 and 64 processes per node.

Table III lists the number of task mappings that were generated for each kernel or application at each node count. For example, for 2D Halo, we created 84 different task mappings and ran them for the two message sizes – 16 KB and 4 MB (168 in total). Added across all the communication kernels, we had 1008 different executions (84 per dataset over twelve datasets) and for the two applications, we had 508 executions (in four datasets). Section IV describes the process of splitting the individual datasets into training and testing sets.

C. Learning predictive models

Building a non-parametric regression model from data is a common task in machine learning applications. In theory, a domain expert specifies an appropriate model and its parameters are suitably adjusted based on observed data. However, in practice, we lack the knowledge of an underlying model for real applications. Hence, it is typical to infer models directly from data, which requires that supervised data with the desired target variables be prepared beforehand. A variety of such data-driven modeling algorithms have been proposed in the machine learning literature [15], [16], and these methods provide a single, "strong" predictive model with good generalization characteristics. An alternative approach is to infer an ensemble of relatively "weak" models to obtain a stronger ensemble prediction [17].

Ensemble methods: The primary reasons for considering ensemble models are: (1) *Statistical*: different predictive models may perform similarly on the training data, when learned from a limited number of training samples. However, the performance of each of these models with test data can be poor. By averaging representations obtained from an ensemble, we may obtain an approximation closer to the true test data; (2) *Computational*: even with large training sets, the modeling technique might not reach the global optimum and using an

ensemble of multiple locally optimal models can result in improved performance; (3) *Representational*: the hypothesis space assumed for learning the model cannot represent the test data, and this can happen when the data is corrupted.

In our previous work [10], we adopted one such approach, extremely randomized trees [6], for predicting execution time based on the communication data. This algorithm builds a forest of decision trees using a top-down approach that progressively partitions the input space into regions where the output is constant. Rather than using a bootstrap of samples, this method uses the whole training set and splits nodes by choosing attributes and cut-points at random.

In this paper, we experimented with a broad class of regression techniques, including support vector machines, ridge regression, Bayesian ridge regression, decision trees, and ensemble learning approaches such as random forests and gradient tree boosting. Extensive evaluation (using cross-validation) of the methods on all our datasets showed that extremely randomized trees and gradient boosted regression trees [7] performed consistently better than the other methods. We use the Python-based *scikit-learn* package [18] for our analysis, which provides the `ExtraTreesRegressor` and `GradientBoostingRegressor` classes.

Gradient boosted regression trees: The main idea of boosting is to add a new weak, base-learner model in each iteration, which is trained with respect to the error of the whole ensemble inferred so far. In gradient boosted regression trees (GBRT) [7], the new base-learners are designed to be maximally correlated with the negative gradient of the loss function associated with the whole ensemble. This technique is flexible enough to be used with different families of loss functions, and this choice is often influenced by the desired characteristics of the conditional distribution, such as robustness to outliers. Any arbitrary loss function can be plugged into the framework by specifying the loss function and the function to compute its negative gradient [19]. The squared ℓ_2 loss and the Laplacian ℓ_1 loss are common choices for regression tasks and these functions penalize large deviations from the target outputs, while ignoring smaller residuals. In addition, parameterized loss functions, such as Huber, can be adopted for robust regression. Given the input variable x, the target output y and the regression function f, the Huber loss is defined as

$$\Psi^H(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & |y - f(x)| \leq \delta, \\ \delta(|y - f(x)| - \delta/2) & |y - f(x)| > \delta. \end{cases} \quad (1)$$

As it can be observed, Huber loss combines ℓ_1 and ℓ_2 functions. The parameter δ is the cutting-edge parameter, and this specifies the maximum value of error beyond which the ℓ_1 function is applied. Alternatively, we can predict a conditional quantile of the target variable for robust regression. This can be achieved by considering the asymmetric quantile loss function:

$$\Psi^Q(y, f(x)) = \begin{cases} (1 - \alpha)|y - f(x)| & y - f(x) \leq 0, \\ \alpha|y - f(x)| & y - f(x) > 0. \end{cases} \quad (2)$$

The parameter α specifies the desired quantile of the conditional distribution. When $\alpha = 0.5$, the quantile loss function corresponds to the ℓ_1 loss. Figure 1 illustrates the Huber and quantile loss functions at different parameter values.

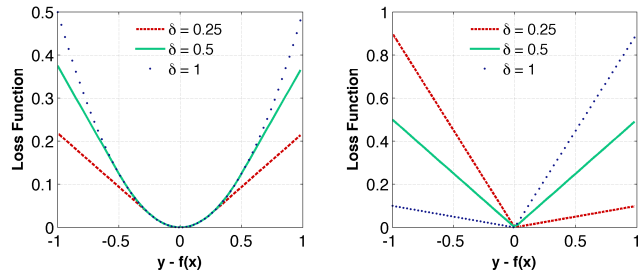

Fig. 1. Parameterized loss functions for gradient tree boosting: Huber loss function with the cutting-edge parameter δ (left), quantile loss function (right)

D. Predicting execution time using different features

There are nineteen features that represent each sample in our input datasets (Table II). Individual features do not lead to good predictions [10], so we need to find the best combination of features for every dataset that leads to the highest prediction scores. An exhaustive search over all possible combinations requires running the boosting algorithm $2^{19} - 1 = 524,287$ times. Each invocation of the algorithm takes about a second, and doing this for even a single dataset would take roughly six days. To overcome this problem, we exploit the ability to run each combination independently. We use pyMPI to divide the work of trying different feature combinations among parallel processes. For these parallel runs, we used Sierra, an Intel Xeon cluster at LLNL. We used 64 nodes (12 cores per node) for each parallel run, bringing the execution time down to under five minutes.

To summarize, these are the steps we follow for learning a model and predicting execution time for an individual dataset:

- Scale each feature in the dataset to have values between 0 and 1 based on the minimum and maximum values for that feature across all samples.
- Divide the n samples in the dataset into a training set and a testing set, roughly in a two-thirds and one-third split.
- Generate all possible combinations of the nineteen features that we would like to learn a model with.
- Do a parallel run where each process runs the GBRT regressor on a subset of feature combinations and reports the prediction scores using the generated models.
- Based on the prediction scores, we pick the feature combinations that lead to the highest scores.

Below, we define the prediction criteria or scores we use to evaluate and compare different learning methods, loss functions and feature combinations.

Evaluation criteria: In order to evaluate and compare different regression models, we consider two different performance scores: the *Kendall rank correlation coefficient* (RCC) [20], and the coefficient of determination, also referred to as the

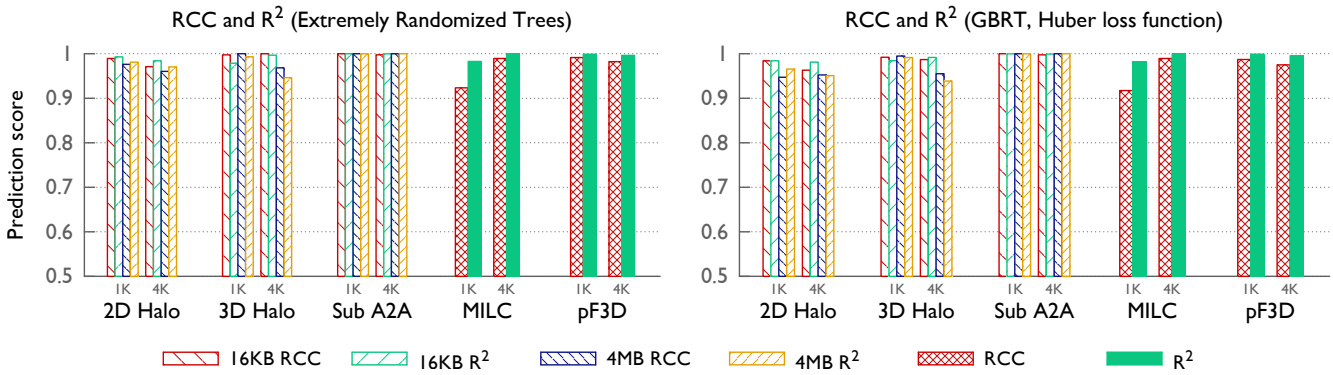

Fig. 2. Highest prediction scores obtained for the individual datasets using Extremely Randomized Trees (left plot) and Gradient Boosted Regression Tree (right plot). Adjoining pairs of vertical bars represent the RCC and R^2 values for each of the sixteen datasets.

R^2 statistic. The RCC score measures the degree of similarity in the rankings of two datasets ($\{x_1, x_2, \cdots, x_n\}$ and $\{y_1, y_2, \cdots, y_n\}$), and can be defined as

$$RCC = \Big(\sum_{0<=i<n} \sum_{0<=j<i} concord_{ij} \Big) / (\frac{n(n-1)}{2})$$

$$concord_{ij} = \begin{cases} 1, & \text{if } x_i >= x_j \ \& \ y_i >= y_j \\ 1, & \text{if } x_i < x_j \ \& \ y_i < y_j \\ 0, & \text{otherwise} \end{cases}$$

The RCC score assumes the value 1 when the two rankings completely agree, while the value 0 indicates complete disagreement. The coefficient of determination is another popular statistic that measures how well a statistical model fits the data.

$$R^2(y, \hat{y}) = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

where $\hat{y}_i$ is the predicted value of the ith sample, y_i is the corresponding true value, and

$$\bar{y} = \frac{1}{n_{samples}} \sum_i y_i$$

denotes the sample mean of the observed data. We use the RCC score to compare the rankings of the true and predicted values in the testing set and R^2 to compare the true and predicted values of the samples directly.

IV. Prediction on Individual Datasets

We begin by using supervised learning to predict the execution time based on communication features, and we evaluate the prediction accuracy. We take the sixteen datasets shown in Table III, and we learn models for each one independently using all possible combinations of input features.

Figure 2 shows the highest prediction scores (both RCC and R^2) obtained for any feature combination for each of the sixteen datasets. Adjoining pairs of vertical bars represent the RCC and R^2 values for each of the sixteen datasets. The left plot illustrates results obtained using the extremely randomized trees algorithm, and the right plot shows similar results using GBRT with the Huber loss function. Though either of the methods can be used for subsequent analysis, we choose GBRT for the results in the rest of the paper because of its flexibility in allowing parameterized loss functions.

The first thing to observe in Figure 2 is that on an average, the prediction scores are very high, which suggests that the communication data and execution time for our datasets are highly correlated. When predicting the execution time of 2D and 3D Halo, we obtain RCCs in the range $0.95 - 1.0$ and R^2 in the range $0.94 - 0.996$. A trend that is not quite discernible from the plots is that, as we increase the amount of communication being performed (from 2D Halo to 3D Halo to Sub A2A), the predictions become stronger. For Sub A2A, the RCC and R^2 values are between 0.997 and 1.0. This is not unexpected – the more a parallel code stresses the network, higher is the correlation between the communication features that represent congestion and execution time.

Even for production applications, which have more complex communication patterns, we observe very high prediction scores. MILC, which performs a 4D halo, is communication-heavy and task mapping sensitive. Other than the RCC scores on 1K nodes, the prediction scores for MILC are very high (R^2 between 0.98 and 0.999). pF3D has communication patterns similar to Sub A2A along with a near-neighbor communication, which results in high RCC values between 0.975 and 0.991. This can be attributed to the structured and communication-intensive all-to-all operations whose execution time is heavily dependent on network congestion.

The prediction scores for pF3D have improved considerably compared to our previous work. On 1K nodes, the R^2 values have improved from 0.93 to 0.995. On 4K nodes, previously, our best RCC scores for pF3D were around 0.75 and R^2 scores were close to 0. Now, both the scores are in the range $0.975 - 0.996$, which is a significant improvement. This is due to the removal of a performance bug in the code, which helps focus the performance on communication properties, and also in part, from the use of an exhaustive search to find the best possible combination of features.

As we compare the prediction quality of the supervised

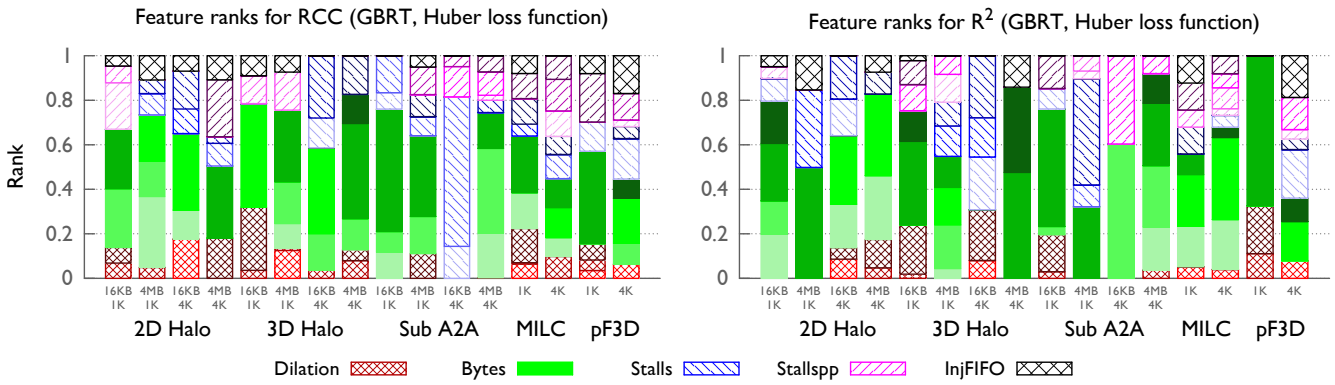

Fig. 3. Ranks of different features in the models that yield the highest RCC (left plot) and R^2 scores (right plot) for individual datasets using Gradient Tree Boosting (loss function = 'Huber'). Each stacked bar represents the ranks of the nineteen features (colored by categories) for one of the sixteen datasets.

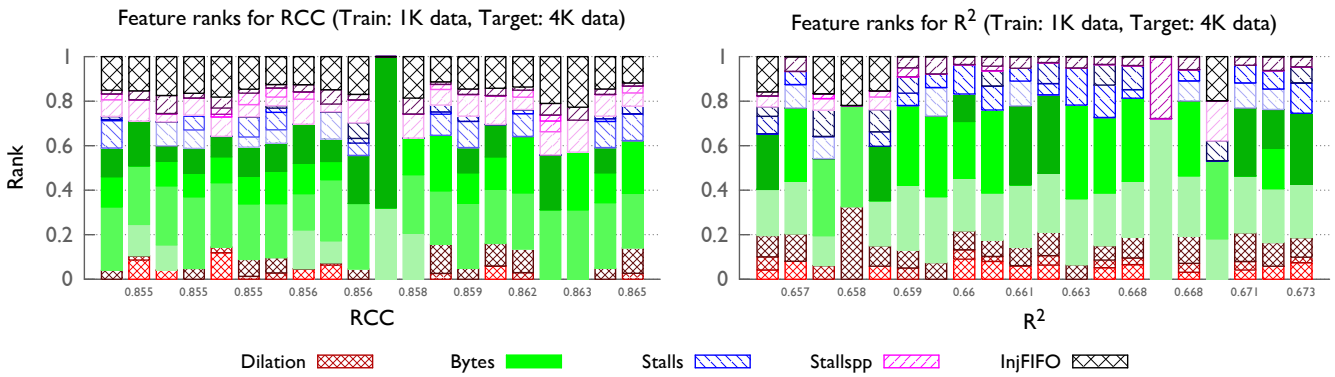

Fig. 4. Ranks of different features for the top twenty RCC (left plot) and R^2 scores (right plot) when using the 1K nodes data to predict the 4K nodes data using GBRT (the top twenty scores are shown in each case using one stacked bar each)

learning models for different codes, a natural question that comes up is – which features are important in predicting the execution time for different kernels and applications? Figure 3 presents the relative importance or ranks of different features in the models that yield the highest RCC (left plot) and R^2 values (right plot). Each stacked bar represents the ranks of the nineteen features (colored by categories) for one of the sixteen datasets. As we can see, the relative importance of features changes depending on the code and on whether RCC or R^2 is being considered. The only conclusive observation that can be drawn from these plots is that the number of bytes flowing over the network has a significant impact on execution time, which is to be expected. Ideally, we would like to identify a smaller subset of features that can predict the execution time well for a range of applications, message sizes and node counts. We discuss this in detail in Section VI.

V. Generalization characteristics

In this section, we analyze the possibility of using the data gathered for a few different codes to predict novel scenarios or applications. The idea is to test if we can apply this approach to predict the execution time in new setups, e.g., on a different number of nodes, or different input datasets, or even for an unknown code – one we have not measured directly yet. We try two different prediction scenarios for understanding the generalization characteristics of our approach.

TABLE IV
Best prediction scores for generalization experiments

Training set	Testing set	RCC	R^2
All 1K samples	All 4K samples	0.865	0.673
All kernels	MILC (1K + 4K)	0.772	0.0
All kernels	pF3D (1K + 4K)	0.874	0.0

In the first scenario, we combine the sixteen datasets by node count into two groups and use the 1K-node data for training and the 4K-node data for testing. In the second scenario, we group the datasets by kernels and production applications, i.e., the twelve datasets for communication kernels combined together are used for training and the four application datasets are used for testing. Table IV shows the best prediction scores we obtain for the different cases by performing an exhaustive search on feature combinations. The network dimensions of the 5D torus and the congestion behavior can be very different on 4K nodes from that on 1K nodes. Even so, the models are able to predict the execution time on 4K nodes reasonably well

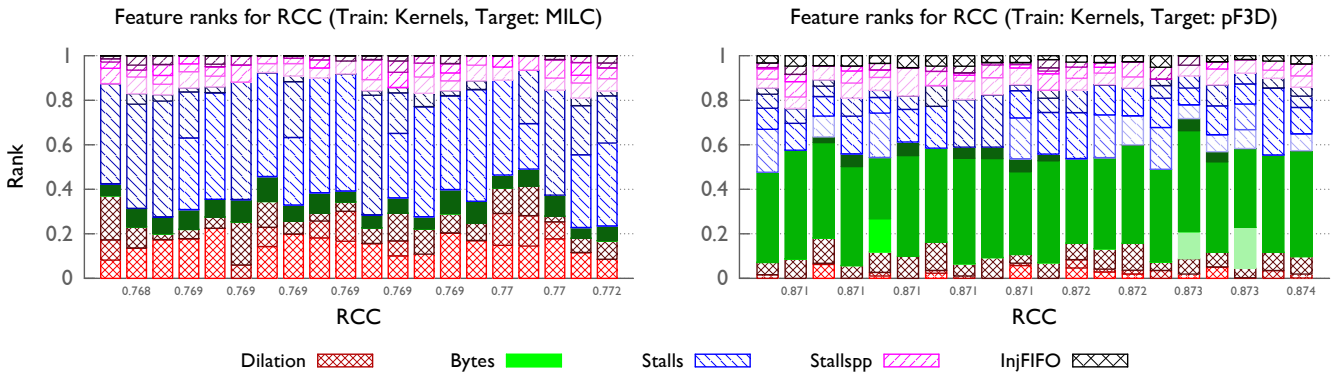

Fig. 5. Ranks of different features for the top twenty RCC scores when using the kernels data to predict the execution time for MILC (left plot) and pF3D (right plot) using GBRT (the top twenty scores are shown in each case using one stacked bar each)

using samples from 1K nodes. We achieve a RCC of 0.865 for the best feature combination and a R^2 value of 0.673. Figure 4 shows the trend of the relative feature importance (rank) as we obtain higher RCC (left plot) and R^2 scores (right plot). Note that feature combinations for the top twenty scores are shown in each case using one stacked bar each. Similar to Figure 3, we can see that bytes on the link is a prominent feature again. Injection FIFO length also shows up as consistently important for high RCC scores (left plot).

The even more interesting scenario is when we use a dataset created from communication kernels to predict production applications. The dataset from all kernels combined together may or may not include samples that represent application behavior. Table IV shows that the RCC value for predicting pF3D's performance is high and that for MILC is also reasonable (0.772). The R^2 scores are 0 when using the communication kernels to predict the execution time for applications. This is an artifact of applying scaling to the datasets before learning the model. Instead of scaling, if we apply standardization on the training and testing sets, we would expect decent R^2 values. The feature importance plots for MILC and pF3D (Figure 5) show that different features are relatively more important for the two applications. In the case of MILC, dilation and stall-based features are more important whereas in the case of pF3D, network bytes is the most important followed by stall-based features. As stated in the previous section, the ideal situation would be to identify a subset of features that yields high correlations for a variety of scenarios.

VI. Identifying relevant feature subsets

The variability in the importance (rank) of different features in the regression models learned for different parallel codes makes it challenging to identify a common set of factors that contribute the most to network congestion. Further, some of the features considered in our analysis might be strongly correlated to one another, thereby introducing instabilities in the model selection process across multiple datasets. In order to overcome these challenges, we propose to infer regression models under different quantiles, and analyze them to identify the most relevant features in a stable manner (irrespective of our choice of training sets). In addition to revealing the hardware components that are the main culprits behind network congestion, this analysis can also provide insights about applications not directly measured and analyzed in this paper.

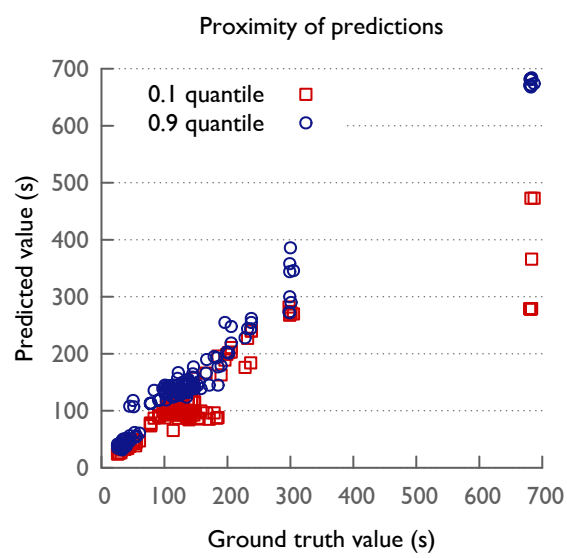

Fig. 6. GBRT regression on the Apps dataset using different quantile loss functions. The lower quantile regression function underpredicts for samples with high execution time, while predicting effectively for those with low execution times.

A. Feature selection from extreme quantiles

For the analysis presented in this section, we use GBRT with the quantile loss function defined in equation (2) in Section III-C. In order to identify the most relevant features for predicting execution time, we propose to analyze the regression models at lower ($\alpha = 0.1$) and higher ($\alpha = 0.9$) conditional quantiles. In particular, we consider the ranks of the different features at the extreme quantiles. Instead of inferring a single regression function that minimizes the

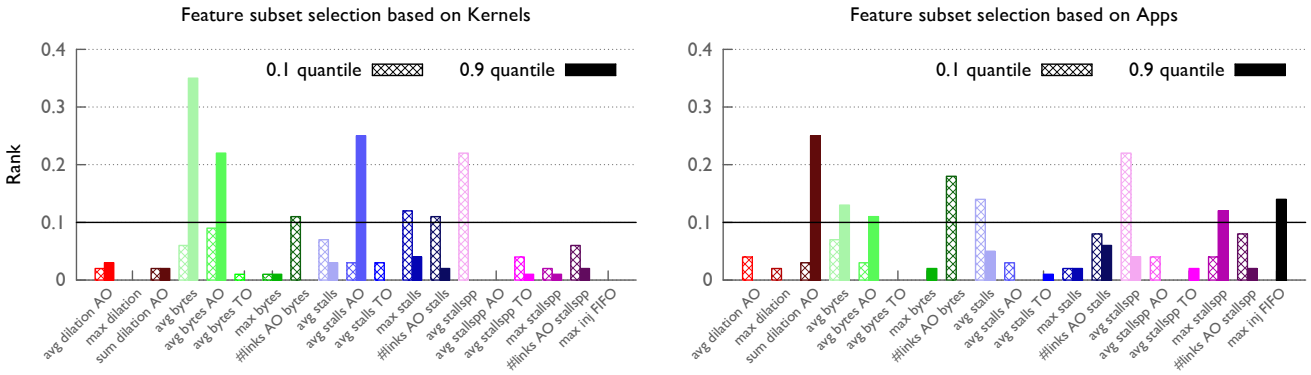

Fig. 7. Ranks of different features obtained using GBRT with quantile loss functions at $\alpha = 0.1$ and $\alpha = 0.9$ respectively: left plot is for a combined set of the three communication kernels (twelve datasets) and the right plot is for a combined set of the two applications (four datasets).

average or median error for all data samples, the quantile loss weights different regions in the function space asymmetrically (see Figure 1). For example, in Figure 6, the lower quantile model provides an accurate prediction for samples with low execution times (bottom left corner), while making large errors on samples with high execution times.

It turns out that for the datasets used in this paper, optimizing for the conditional quantiles inherently promotes sparsity in the inferred model (Figure 7). This means that only a few features show significant importance for prediction, and the ranks of different features vary considerably in the case of lower versus higher quantiles. This results in different features being more important for the two quantiles. Figure 7 shows the feature importance for the extreme quantiles for all the kernel datasets combined together (left plot) and all the application datasets combined together (right plot). In the left plot, we see that the features *avg bytes* and *avg stalls AO* have a high rank only when predicting at the higher quantile. On the other hand, the feature *avg stallspp* is prominent in predicting at the lower quantile and not used by the regression function optimized for the higher quantile. In the right plot, we can observe similar things about *sum dilation AO*, *max inj FIFO*, *#links AO bytes* and *avg stallspp*.

We exploit these observations by selecting the most relevant features from the models at different quantiles, and using this subset of features to predict the execution time for different applications. The steps involved in this proposed technique for feature selection for a dataset are as follows:

- Create random splits of the dataset into training and testing sets (70% for training and the rest for testing).
- Learn regression models using GBRT with quantile loss functions at $\alpha = 0.1$ and $\alpha = 0.9$. We denote the feature ranks in the two cases by $\tau_{0.1}$ and $\tau_{0.9}$ respectively.
- Repeat the above steps 50 times to avoid overfitting and compute the average feature ranks for the extreme quantiles from the 50 iterations.
- Identify the relevant features as those with either $\tau_{0.1}$ or $\tau_{0.9}$ greater than a pre-defined threshold t. In our experiments, we fixed t at 0.1.

B. Results and discussion

We employ the feature selection technique explained above on the following larger datasets formed by combining the individual datasets in Table III:

1) 2D Halo (4 datasets)
2) 3D Halo (4 datasets)
3) Sub A2A (4 datasets)
4) Kernels (combination of (1), (2), and (3), 12 datasets)
5) MILC (2 datasets)
6) pF3D (2 datasets)
7) Apps (combination of (5) and (6), 4 datasets)
8) All (all 16 datasets added together)

The goal is to identify a common set of features that might be relevant across multiple datasets. Figure 8 presents the feature ranks obtained using the technique described above for each of the larger datasets. Note that the importance/rank of each feature is obtained by first identifying the smallest subset of important features for each dataset and then performing another cycle of training and testing to obtain the relative importance of the features in this identified subset. The marker colors for each row/dataset are scaled independently (maroon/red is high and yellow is low).

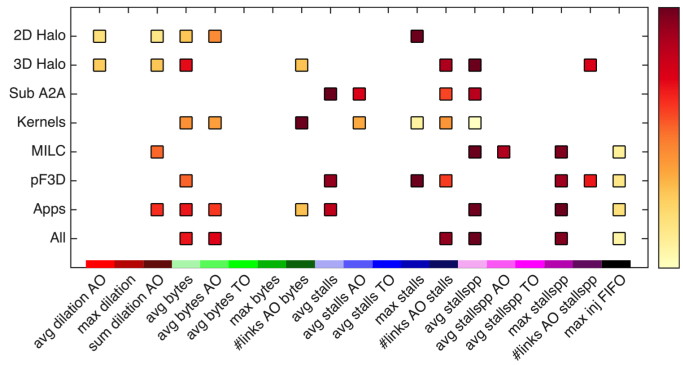

Fig. 8. Comparison of the feature ranks obtained using the feature selection technique applied to the eight larger datasets. Note that the marker colors for each row are scaled independently (maroon/red is high and yellow is low).

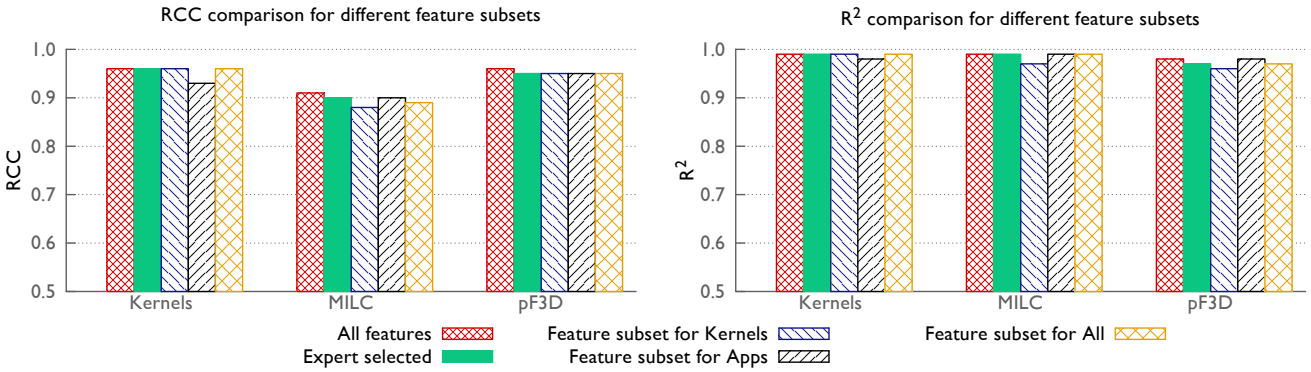

Fig. 9. Prediction performance of the features selected using the proposed quantile analysis on different datasets. RCC and R^2 values are averages over 50 random splits of the datasets into training and testing sets.

We can make a few important observations from Figure 8. The markers in the row for the *All* dataset show that the "stalls" features are the most important. The stalls group indicates scenarios in which a network packet has to wait in the receive buffer. The wait could either be on an intermediate node because the next link is busy or on the destination node because the node is not able to consume the received packets at the same rate as they arrive. *Stallspp* refers to the number of stalls encountered on a link per packet. The high ranks of these features suggest that the receive buffers on the nodes are one of the most important causes of network congestion.

The other important feature in the *All* dataset of Figure 8 is *avg bytes*. This refers to the average number of bytes passing through a network link and is an indicator of the average traffic on the network. A high rank for this feature suggests that to mitigate congestion, algorithms and task mappings should aim to minimize the average load per link. It also indicates that *max bytes* or the most overloaded link (often referred to as a hot-spot) is not a strong determinant of the execution time. Finally, we note that *max inj FIFO* or the maximum injection FIFO length also plays a small role in predicting the execution time, especially for the production applications.

We can also observe which features are important for a particular larger dataset using Figure 8. For example, *max inj FIFO* is only important for the production applications (MILC and pF3D) and the Apps and All datasets. *Avg stallspp* is important across most datasets but *max stallspp* is only important for the production applications and their combinations. *Avg bytes* is also important in almost all datasets except Sub A2A and MILC. Further, it is important to note that a feature might not show up as important for a dataset in this figure for one of the following two reasons – either it was less important than the top five to seven features or another feature that highly correlates with this feature was in the top list.

In order to evaluate the performance of the relevant feature subsets obtained using feature selection, we learn regression functions using the subsets on the following datasets: Kernels, MILC, and pF3D. In addition, we compare the performance for these feature subsets with that obtained by using all nineteen features and an expert selected subset of twelve features. For the expert selection, we only pick those features that we believe represent some unique information about the dataset. We pick one of two features if they are known to be highly correlated. In each case, we run the GBRT algorithm with 70% of the data for training and the rest for testing. The results reported in Figure 9 are obtained by calculating average RCC and R^2 scores over 50 random splits of the datasets into training and testing sets.

In Figure 9, we observe that if we use the feature subset from the communication kernels to predict MILC or pF3D, there is a performance drop. This is also true if we use the feature subset from the *Apps* to predict the communication kernels. This suggests that the communication kernels dataset has some characteristics that are not well modeled by the features extracted from the *Apps* dataset and vice versa.

Nonetheless, the important result in Figure 9 is that using a feature subset derived from all the datasets, we can do reasonably good predictions for communication kernels and production applications. These predictions are close to predictions obtained using all nineteen features. This subset of features is: *avg bytes, avg bytes AO, #links AO stall, avg stallspp, max stallspp and max inj FIFO*. From these results, we conclude that the features that are among the primary root causes of network congestion are (in decreasing order of importance): the average and maximum lengths of receive buffers, average load on the network links, and the maximum length of injection FIFOs on the source node. We also observe that the maximum load on a link (network hot-spots) and message dilation or the number of hops a message travels are lesser indicators of network congestion.

VII. Summary

The ability to predict the performance of communication-heavy parallel applications without actual execution can be very useful. This requires understanding which network hardware components affect communication and in turn, performance on different interconnection architectures. A better understanding of the network behavior and congestion can help

in performance tuning through the development of congestion-avoiding and congestion-minimizing algorithms.

This paper presented a machine learning approach to understand network congestion on supercomputer networks. We used regression analysis on communication data and execution time to find correlations between the two and to learn models for predicting execution time of new samples. Using a quantile analysis technique to identify relevant feature subsets, we were also able to extract the relative importance of different features and that of the corresponding hardware components in predicting execution time. This helped us to identify the primary root causes or culprits behind network congestion, which is a difficult challenge.

Using our methodology, we obtained prediction scores close to 1.0 for individual datasets. We were also able to reasonably predict the execution time on higher node counts using training data for smaller node counts. We also obtained reasonable ranking predictions for new applications using datasets based on communication kernels only. Finally, we identified the hardware components that are primarily responsible for impacting and hence, predicting the execution time. These are – receive buffers on intermediate nodes, network links and injection FIFOs in decreasing order of importance. We also observed that network hot-spots and dilation or the number of hops a message travels are lesser indicators of network congestion. This knowledge gives us a real insight into network congestion on torus interconnects and can be very useful to network designers and application developers.

Finally, the prediction techniques presented in this paper are widely applicable to a variety of scenarios, such as, (1) creating offline prediction models that can be used for low overhead tuning decisions to find the best configuration parameters, (2) predicting the execution time in new setups, e.g., on a different number of nodes, or different input datasets, or even for an unknown code, (3) identifying the root causes of network congestion on different architectures, and (4) generating task mappings for good performance.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by the Laboratory Directed Research and Development (LDRD) Program at LLNL under project tracking code 13-ERD-055 (LLNL-CONF-663150). This research used computer time on Livermore Computing's high performance computing resources, provided under the LDRD Program.

References

[1] C. Hyatt and D. P. Agrawal, "Congestion control in the wormhole-routed torus with clustering and delayed deflection," in *Parallel Computer Routing and Communication*, ser. Lecture Notes in Computer Science, S. Yalamanchili and J. Duato, Eds. Springer Berlin Heidelberg, 1998, vol. 1417, pp. 33–38.

[2] A. Bhatele and L. V. Kale, "Quantifying network contention on large parallel machines," *Parallel Processing Letters*, vol. 19, no. 04, pp. 553–572, Dec. 2009. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626409000419

[3] J. Escudero-Sahuquillo, E. Gran, P. Garcia, J. Flich, T. Skeie, O. Lysne, F. Quiles, and J. Duato, "Combining congested-flow isolation and injection throttling in hpc interconnection networks," in *2011 International Conference on Parallel Processing (ICPP)*, Sept 2011, pp. 662–672.

[4] A. Bhatele, E. Bohm, and L. V. Kale, "Optimizing communication for Charm++ applications by reducing network contention," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 211–222, Feb. 2011. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/cpe.1637/abstract

[5] 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.

[6] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[7] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[8] N. Sukhija, B. Malone, S. Srivastava, I. Banicescu, and F. M. Ciorba, "A learning-based selection for portfolio scheduling of scientific applications on heterogeneous computing systems," *Parallel and Cloud Computing*, vol. 3, no. 4, pp. 66–81, Oct. 2014.

[9] N. R. Tallent, A. Vishnu, H. Van Dam, J. Daily, D. J. Kerbyson, and A. Hoisie, "Diagnosing the causes and severity of one-sided message contention," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. ACM, 2015, pp. 130–139. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688516

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

[11] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[12] 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.

[13] S. Langer, A. Bhatele, and C. H. Still, "pF3D simulations of laser-plasma interactions in National Ignition Facility experiments," *Computing in Science and Engineering*, vol. 99, Aug. 2014, lLNL-JRNL-648736. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MCSE.2014.79

[14] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, "Mapping applications with collectives over sub-communicators on torus networks," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, Nov. 2012, LLNL-CONF-556491.

[15] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani, *The elements of statistical learning*. Springer, 2009, vol. 2, no. 1.

[16] C. M. Bishop *et al.*, *Pattern recognition and machine learning*. springer New York, 2006, vol. 1.

[17] T. G. Dietterich, "An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization," *Machine learning*, vol. 40, no. 2, pp. 139–157, 2000.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, 2013.

[20] Kendall tau rank correlation coefficient. http://en.wikipedia.org/wiki/Kendall_tau_rank_correlation_coefficient.