Identifying HPC Codes via Performance Logs and Machine Learning

Orianna DeMasi
Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley, CA 94720
ODeMasi@lbl.gov

Taghrid Samak
Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley, CA 94720
TSamak@lbl.gov

David H. Bailey
Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley, CA 94720
DHBailey@lbl.gov

ABSTRACT

Extensive previous work has shown the existence of structured patterns in the performance logs of high performance codes. We look at how distinctive these patterns are and question if we can identify what code executed simply by looking at a performance log of the code. The ability to identify a code by its performance log is useful for specializing HPC security systems and for identifying optimizations that can be ported from one code to another, similar, code. Here we use supervised machine learning on an extensive set of data of real user runs from a high performance computing center. We employ and modify a rule ensemble method to predict what code was run given a performance log. The method achieves greater than 93% accuracy. When we modify the method to allow an "other class," accuracy increases to greater than 97%. This modification allows an anomalous run to be flagged as not belonging to a previously seen, or acceptable, code and offers a plausible way to implement our method for HPC security and monitoring what is run on supercomputing facilities. We conclude by interpreting the resulting rule model, as it tells us which components of a code are most distinctive and useful for identification.

Keywords

Distributed applications, Network monitoring, Performance Modeling and prediction, Machine learning

1. INTRODUCTION

Monitoring the performance of applications running on high performance computing (HPC) resources provides great insight into application behavior. It is also necessary for monitoring how HPC resources are used. Competitive allocation of HPC resources for users and the expense of maintaining HPC centers demands that tools be available to identify whether a user is running the approved code and whether the system is being used as intended.

Most performance monitoring tools provide access to offline logs that describe details about individual application runs. For systems as big as National Energy Research Scientific Computing Center (NERSC), the process of inspecting and analyzing those logs is very difficult, considering the massive dataset generated across applications. Automatic analysis of application performance logs presents the only way to accommodate the ever-increasing number of users and applications running on those large-scale systems. The ability to identify a code from system logs would enable accurate anomaly detection on both code and user levels. Automatic code identification would also improve understanding of application behavior, which would lead to better resource allocation and provisioning on HPC resources, as well as better auto-tuning and improving the performance of a given code.

Previous work has examined a variety of high performance scientific codes and found striking patterns in the communication behavior of each code [10, 14, 16, 17]. However, these case studies have not been able to address the uniqueness or identifiability of a code's behavior, disparity from the behavior of other similar computations, or robustness of behavior to method parameters, problem size, and the number of nodes the code was run on. Algorithms scale differently, and these studies have considered this behavior from a performance perspective, so they did not determine if one code will scale differently and lose its ability to be identified. Further work has argued that codes can be grouped into several groups, or dwarves [3, 11], of computation, where codes within a group have features that are common to all codes in the group. The features of a code described by the paradigm of computational dwarves, as well as the empirical results showing structured communication patterns, inspire the question of whether code traces are unique enough to permit identification by their performance log. The dwarves system of classification indicates that codes should be identifiable and that uniquely predicting codes could be possible.

While these studies have examined communication patterns and postulated that communication patterns are characteristic to codes, there has been little effort to formalize or automate this concept and test a large set of codes and runs made by users. Previous studies have used very controlled sets of codes, and have chosen codes that are from very different computational families, which would be expected to have very different behavior. Further, previous studies have used fairly small datasets that make generalizing arguments

to the complex terrain of all high performance computing very difficult.

In this paper, we present a machine learning based approach to enable large-scale analysis of HPC performance logs. Machine learning has emerged as an extremely powerful means for analyzing and understanding large datasets. Supervised learning algorithms enable accurate classification of data patterns, and separation of classes (in our case, codes). They also provide great insights in understanding patterns and attribute interactions for the identified classes. We aim here to leverage supervised learning to enable large-scale analysis of performance logs, in order to accurately classify code runs and understand the importance of different performance metrics.

This study extends previous work by looking at a broad range of codes and a much larger set than previous studies have been able to consider. Our set of observations includes codes that are similar in nature, e.g. multiple linear algebra codes, and that are not as easily distinguished from each other than codes that perform very different computations, e.g. a particle in cell code instead of a linear algebra code. This study also differs from previous studies in that we use profiles of code runs made by users on a large high-performance computer system facility, namely the NERSC at the Lawrence Berkeley National Laboratory. The observations used are not just benchmark codes or multiple runs using a single functionality of a single code, but are representative of the complex computations that are routinely executed at super computing facilities.

Our contribution can be summarized in the following key points:

- We applied the Rule Ensemble classification algorithm to accurately classify scientific codes running at NERSC.
- We extended the Rule Ensemble method to handle multi-class classification problems, and account for unknown classes.
- The extended method is applied to a large set of performance logs collected at NERSC.
- We performed rigorous attribute analysis for performance metrics with respect to code classes.

In Section 2, we describe how we collected Integrated Performance Monitoring (IPM) performance logs from a broad set of applications at the NERSC facility. In sections 3 and 4, we describe the supervised learning method that we used, as well as the application specific alterations and our experimental setup. Section 5 presents our results of how accurately we are able to identify codes. This section also discusses the accuracy when the classification is relaxed and observations are allowed to be "unclassified" or flagged for further reference. Section 6 interprets the models that were built and considers what this tells us about the data. Section 8 discusses the results and Sections 7 – Section 9 present related work and future directions.

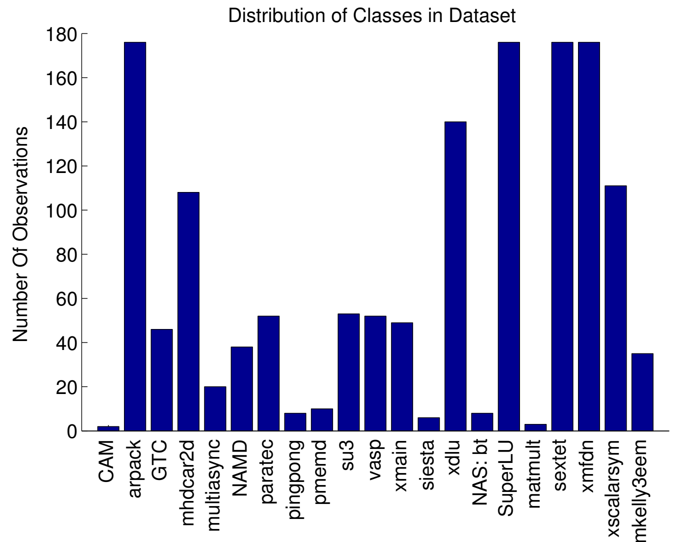

Figure 1: The number of observations of each class that are used in the training set and the fraction of the total dataset that class contributes.

2. DATA COLLECTION, PROCESSING, AND FEATURE EXTRACTION

One significant contribution of this work is the size, quality, and breadth of the dataset that we explore. A variety of papers have looked at similar types of data [10, 14, 16, 17], but we are able to present a dataset that was generated in an uncontrolled environment and thus is more representative of real workloads on supercomputing facilities. Our dataset also has many more observations and more applications than previous sets, which allows for better inference to behavior beyond the scope of our dataset. In this section we describe how our data was collected and preprocessed, as well as its significance and extension of previous work.

2.1 Data Format

The dataset used consists of 1,445 performance profiles or logs of highly parallel codes executed on systems at the NERSC facility. The codes profiled are for scientific applications and are regularly run by users on of the NERSC computers and thus are representative of workloads on NERSC. The codes and number of observations of each code are listed in Figure 1.

A subset of the logs was generated by users and stored by NERSC for research. The remaining logs were generated by NERSC staff for benchmarking purposes. These logs represent a suite of benchmarks that are regularly run to maintain and check the performance of the systems. The benchmark suite represents a broad range of scientific applications and computations that are run on the machines.

The performance of codes was recorded with the Integrated Performance Monitoring (IPM) software tool [2, 15], which logs the execution of a code in an XML file. IPM is a relatively lightweight performance tool that can provide various levels of performance data. We chose to use IPM, rather

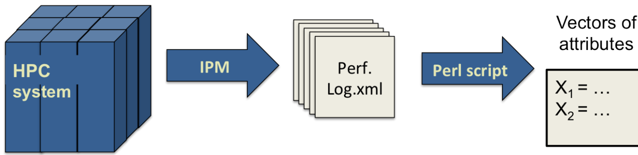

Figure 2: Diagram showing the path of data being converted from IPM output XML files to data vectors that can be fed to learning algorithms.

than another profiling tool, because it has very little overhead (less than 5% slow down [1]), is extremely easy to use, and captures fine grain node-to-node communication that other profiling tools do not capture. IPM only requires that the code be linked to the IPM library at compile time and the granularity of the profile must be declared at compile time.

The process used to convert each XML log into a vector of data that could be fed as an observation to a machine learning algorithm is depicted in Figure 2. Each XML log contains a profile of a given code execution. We used the parser that is distributed with IPM to generate a "full" IPM profile. The full profile is a list of high level statistics that indicates how a code spent its time and is the default level of output that IPM delivers at the end of an execution. The information summarized in a "full" profile is collected at the least invasive setting of IPM. Even though more specific and detailed information was included in many of the logs that we had access to, we wanted to establish if the most basic level of information about a code run was sufficient to identify which code was run.

After logging a code and collecting the "full" IPM profile, we used a python script to extract a vector of features from the "full" profile that describe the code run. The features that were extracted are described in the following section. Ideally, each vector of features will allow a trained model to predict what code was run.

2.2 Features for Supervised Learning

To understand and correctly classify codes, we need a list of features that can represent and fully capture the meaningful and unique behavior of a code. The main features that we want to capture are load balancing, or how the code distributes work and data to nodes, computational complexity, how the code is communication, how much the code is communicating, and what processes the code spends its time on. Because each code can be run on a different number of nodes, we consider the average timings for each node and the range or variance of timings between nodes. For organization, we break the measurements into three major types: timing data, communication measurements, MPI data. Note that the measures used exist on vastly different scales. Some measures are percents of time, which are in the interval $[0, 1]$, and others are the total number of times that a call was made during the execution of a code, which can be in the range $[0, +300, 000]$. The huge discrepancy between scales is one source of difficulty in understanding and modeling this dataset.

2.2.1 *Timing data*

To quantify how each code spent its time and the uniformity of nodes, we looked at the percent of the total time that was spent in user, system, and mpi calls. We also looked at the range for each of these measures between the nodes by taking the difference of the maximum and minimum amount of time a node spent in one of these sectors and dividing by the average time that a node spent.

% time in user calls	% time in system calls	% time in user calls	%of wall time in sends

Table 1: Subset of the measures used to quantify how time was spent.

The amount of time that a node spent in communication was captured by a feature representing the average percent of the wallclock time that a node spent executing mpi commands. Another feature was the range of the percent of wallclock time that a node spent executing mpi commands.

To measure data flow and computation rate, we considered the total gigaflops/second that the application maintained (this is the sum of all the flops on all of the nodes) as the range of gflops/sec between nodes. We did not track the total data flow as from preliminary tests we found that the total data that must be moved is altered by the size of the problem and the data that one inputs to a code. As this quantity varies more by how the code is used, we only use the range of total gigabytes of data that each node uses. This measure of range is valuable, as it yields a sense of how imbalanced data movement is in the code is and whether data is equally distributed between nodes. An example subset of features in this type of data is given in Table 1.

2.2.2 *Communication measurements*

To quantify the general behavior of the code, we want describe how the code spends the majority of its time. Some computations are very fine grain and perform mostly point to point communication with one processor sending to another. Other computations are much more global is scope and spend the majority of time doing global updates will all the nodes communicating with all the other nodes.

time in {blocking}	# {blocking} calls	% mpi time in {blocking}	% wall time in {blocking}

Table 2: By {blocking} we mean all blocking communication. This subset of measures was used to quantify the blocking communication of a code. Similar measures were computed for all the subsets of communication that are listed in Table 2.3.

To capture the general communication behavior, we group commands into a variety of lists: all commands, blocking commands, nonblocking commands, sending calls, receiving calls, point to point calls, collective calls, other calls. The exact commands that are included in each list are discussed in the appendix 2.3. For each of these lists we calculate the total time spent in the commands in each list, the number of calls that were made from the commands in each list, the percent of mpi time that the commands in each list accounted for, and the percent of wall time that was spent

executing commands from each group. An example subset of features in this type of data is given in Table 3.

time in sends	# of sends	% of mpi time in sends	%of wall time in sends

Table 3: Subset of measures used to quantify the global communication of a code.Similar measures were computed for all the subsets of communication that are listed in Table 2.3.

2.2.3 MPI data

To understand how the nodes communicated within the run of a code, we look at which MPI commands were used, and the burden of communication that they bear. For each MPI command, we consider the total time that was spent executing that command, the number of times that it was called, the percent of the total communication time was spent in that call, and the total percent of the wall time that the code represented. Different MPI commands are used by different codes. The choice of which commands are used can help to identify codes. If a certain command was not used, then the values for time, number of calls, percent of MPI time, and percent of wall clock time for that command are set to zero. This method can cause problems as commands that are not widely used by many commands will have little support, or have nonzero entries for few observations. An example subset of features in this type of data is given in Table 4.

time calling MPI_Send	# MPI_Send calls	% of comm. in MPI_Send	% wall time in MPI_Send

Table 4: Subset of the measures used to quantify how the code used the MPI library and what commands nodes use to communicate.

2.3 Challenges with data

Collecting a dataset large enough to represent the population of codes on NERSC was difficult. One of the challenges faced was understanding what codes were used without having explicit labels on the XML logs. Once logs were collected, it was not clear what the input data and method parameters were for the given run. Not knowing the input and parameter settings made it difficult to know how much of the possible variation was represented in the collected data.

The collection of codes was imbalanced in the number of observations collected for each class. Certain classes contained many orders of magnitude more observations than other codes, which had very few observations. Especially for supervised learning algorithms, this imbalanced caused many problems, as one class dominated the model. To mitigate this problem, we down selected over-represented classes by taking a random subset of observations from the given class and removing the rest. We did this to the few classes that far exceeded the other classes in the number of observations collected.

Another challenge we faced was deciding which features to extract from the XML logs or from the "full" profiles. Feature extraction is a difficult topic for any application of machine learning and is usually based on domain expertise.

sending	`MPI_Send, MPI_Rsend, MPI_Isend`
receiving	`MPI_Recv, MPI_Irecv`
collective	`MPI_Allgatherv, MPI_Allreduce, MPI_Alltoallv MPI_Barrier, MPI_Bcast, MPI_Gather MPI_Gatherv, MPI_Reduce, MPI_Testall MPI_Waitall, MPI_Waitany`
other	`MPI_Iprobe, MPI_Test, MPI_Barrier`

Table 5: MPI commands that were measured and used to describe the behavior of codes.

Here we selected features which we felt represented the general trends of where a code spent its time. We tried to capture this behavior in a variety of features, many of which were on very different scales. Some features were on scales of $[0, 1]$, such as the fraction of time spent in a certain command, while others were on scales of 10,000's, such as the total number of calls made. To mitigate the effect of features on large scales drowning out features on smaller scales, we normalized the data to have unit variance.

MPI Commands MPI has a large number of commands. However, some are not very descriptive of the performance of a log. For example, `MPI_Init` and `MPI_Finalize` are required to be called to initialize and finalize the MPI environment, but give no information about the performance of the code. We reduce extraneous data by not considering such commands. We focus on a very common subset of MPI commands that are predominately used in practice and account for the large majority of message passing that we expect to see in large scientific codes. The commands that we consider are listed in Table 2.3 and are grouped as sending, receiving, collective, and other calls. We used additional groupings of some of the commands listed in Table 2.3 to develop meaningful statistics. The additional groups that we used were of point-to-point communication, point-to-point blocking calls, and non-blocking point-to-point calls.

3. METHOD: RULE BASED ENSEMBLE

For modeling the data we turn to supervised ensemble methods from machine learning. Ensemble methods have recently become very popular because they are flexible and have high predictive accuracy in nonlinear problems. Ensemble methods are built by combining many simple base learners into a larger model. Each individual base learner could have low predictive capability, but together they capture complex behavior. In addition to powerful models, each base learner is quick to construct and thus ensemble methods tend to be relatively fast. We consider a model that was originally proposed by Friedman and Popescu [6, 8] because in addition to high predictive accuracy it yields a model that can be interpreted. Interpretability is not possible with many other ensemble methods. We begin this section by giving a brief overview of how the method was originally proposed and then describe the extensions and alterations we made to apply the method in this application.

3.1 Overview of Rule Ensemble Method

In the rule ensemble method proposed by Friedman and Popescu [6, 8], the base learners take the form of binary rules that determine if an observation is or is not a given class, denoted by +1 and -1 respectively. The rule ensemble

method considers a set observations $\mathbf{x}_i, i = 1 \dots N$ that have corresponding labels $y_i \in \{-1, 1\}$. The method predicts a label $\hat{y}(\mathbf{x})$ for a previously unseen observation $\mathbf{x}$ by assuming a model that is a linear combination of rules

$$F(\mathbf{x}) = a_0 + \sum_{k=1}^{K} a_k r_k(\mathbf{x}). \quad (1)$$

The label is predicted with

$$\hat{y}(\mathbf{x}) = \text{sign}(F(\mathbf{x})). \quad (2)$$

Each rule r_k is defined by a hypercube in parameter space and is of the form

$$r_k(\mathbf{x}_i) = \prod_j I(x_{ij} \in p_{kj}). \quad (3)$$

Here, p_{kj} defines a region of parameter space and $I(\cdot)$ is an indicator function that returns 1 if the observation does fall into that region of parameter space and 0 if it does not. Each rule indicates if an observation does or does not fall in a certain region p_{kj} of parameter space. For example, a rule might check if a code does or does not use a certain MPI call, say `MPI_Send`. This rule would return 1 if the observed code did use `MPI_Send` and 0 if it did not. Another rule could check if the code used `MPI_send` and spent equal or more than 50% of the total time in system calls (the rule would return 1) or less than 50% of its time in system calls (the rule would return 0).

The rules are found by fitting the p_{kj}'s. These parameter regions are fit by building a series of small decision trees and taking the internal and terminal nodes of each tree as a rule. Each rule r_k is given a prediction weight a_k to control how much it contributes to the final prediction. The rule weights $\mathbf{a}$ are found by a penalized regression

$$\hat{\mathbf{a}} = \arg\min_{\{\mathbf{a}\}} \sum_{i=1}^{N} \left(y_i - a_0 - \sum_{k=1}^{K} a_k r_k(\mathbf{x}_i) \right)^2 + \lambda \sum_{k=1}^{K} |a_k|. \quad (4)$$

The ℓ_1 lasso penalty is controlled by the scalar λ, which determines how much of a penalty is added to the right hand side of Equation 4 for increasing a coefficient. The lasso penalty is used to control how many rules are included in the model and effectively removes excessive rules that have little to no predictive capability [5, 6]. The lasso penalty encourages a sparse solution to the coefficient vector $\mathbf{a}$, or as few rules to be included in the model as possible.

All the rules r_k and weights a_k together form an ensemble that is used to predict which class an observation belongs to. The rule ensemble method has a variety of advantages over other methods. Using the penalized regression removes rules that are not vital to prediction and thus allows for a simpler, more interpretable model. The form of the rules is of particular advantage to our application. Because the rules have binary cuts, they can capture intuitive rules, such as "this code does or does not use `MPI_Send`".

In the present application, there is a large disparity between the number of observations in each class. Having a large imbalance of positive to negative observations makes it difficult to "learn" what a given class looks like. To mitigate this effect, we adjust the threshold using

$$t^* = \arg\min_{-\infty \le t \le \infty} E_{\mathbf{x}y} \left[\frac{(1+y) I(F(\mathbf{x}) \le t) + (1-y) I(F(\mathbf{x}) > t)}{2} \right]$$

where $E_{\mathbf{x}y}$ is the expectation operator and $I(\cdot)$ is an indicator function [8]. We then make label predications with $\hat{y} = \text{sign}(F(\mathbf{x}) - t^*)$. This threshold adjustment allows us to shift the model so that the misclassification error will be minimal on the training set. We considered alternative methods for compensating for the class imbalance, but the results we got by shifting the threshold were sufficiently improved from initial experiments.

We use a variety of modifications to the rule ensemble method that have improved the performance on previous datasets and allowed the method to be used on multi-class datasets [4]. One of these modifications is using a *fixed point continuation method* (FPC) [9] to approximate the solution to Equation 4 instead of the *constrained gradient descent method* (CGD)[7] that was originally suggested for the rule ensemble [8]. The FPC method was found to prune more rules than CGD and thus return a smaller model without sacrificing any accuracy [4].

Rule and Attribute Importance

The relative importance of a rule r_k is measured by

$$I_k = |\hat{a}_k| \cdot \sqrt{s_k(1 - s_k)},$$

where s_k is the rule support on the training data

$$s_k = \frac{1}{N} \sum_{i=1}^{N} r_k(\mathbf{x}_i).$$

This measure takes into account how often a rule is used and how large of a weight it has in the model [6, 8].

The relative importance of the jth attribute is measured with

$$J_j = \sum_{\{k, x^j \in r_k\}} \frac{I_k}{m_k},$$

where m_k is the number of attributes that participate in rule k and the summation is over the rules that consider attribute j. This measure considers how many rules an attribute contributes to and how many attributes is used for each of those rules.

These measures I_k and J_j give relative values; they allow us to say that one attribute or rule is more important than another, but don't yield standard units.

3.2 Extending to Multiple Classes

Another modification uses a One Verses All (OVA) type scheme [13] to extend the rule ensemble to a problem of multiples classes. OVA classification schemes use a binary classification method to decide if an observation is part of one class, or any other class. This check is repeated for each class and a binary valued vector is produced. To avoid predicting that an observation is in more than one class or in no classes, we construct a vector of approximation values. The approximation value $F_j(\mathbf{x}_i)$ is the value resulting from Equation 1 in the model that predicts if an observation is

or is not a member of class j. The final predicted label for an observation $\mathbf{x}_i$ is $\hat{y} = j^*$, where $F_{j^*}(\mathbf{x}_i) > F_j(\mathbf{x}_i)$ for any other j. This method predicts $\mathbf{x}_i$ to be in the class for which the rule ensemble is, in a sense, "the most sure" that it belongs to that class, or the class for which $F_j(\mathbf{x}_i)$ is furthest from the classification margin.

3.3 Extending to Unknown Classes

A potential limitation to applying the above method for security is that it cannot account for observations that are part of a new, unforeseen, class or code. The above extension does not allow for an "other" class where a new code or particularly anomalous observations, which do not appear to fall into any of the previously seen categories, can be classified. This issue is a familiar shortfall of many methods that try to classify new observations as instances of one of the classes that the method was trained on and had already seen. This shortfall is vital to security where it is the anomalous instances that are sought.

To address this problem, we consider that if an observation is not from one of the training classes, then the approximations should be negative for each class; i.e. $F_j(\mathbf{x}) < 0$ for every class j. Above, we made a classification by choosing the prediction $\hat{y}(\mathbf{x})$ to be the class j where the approximation $F_j(\mathbf{x})$ was largest. Now we allow observations to be considered "unclassified", when $F_j(\mathbf{x}) < 0$ for every class j. Relaxing the classification in this manner, allows for attention to be drawn to observations that do not appear to be members of any class. In practice, any observation left "unclassified" could be flagged for further review by a systems expert.

3.4 Extending Attribute Importance to Multi-Class Model

The measures described above are designed to give the relative importance of attributes and rules in each binary model. To get the importance for an attribute in the the multi-class setting, we aggregate the relative rankings with

$$J_j = \sum_{l=1}^{C} (J_j^l)^2.$$

Here C is the number of classes and J_j^l is the relative importance of the jth attribute in the model that tries to identify that lth class. By definition, the relative measures are no larger than one. Squaring the terms in the sum suppresses smaller measures, so that the distribution is more differentiated and it's easier to see which attributes were consistently important. We aggregate measures in this way so that sets of attributes that are very important for identifying one class and sets that somewhat important in many classes both maintain some importance in the global measure.

4. EXPERIMENTAL SETUP

To asses the accuracy of a model we use 5-fold cross-validation. In this process the data is split into five subsets that have roughly the same number of observations and preserve the class distribution. A model is then trained on four of the subsets and tested on the fifth. This process is repeated five times, each time training on a different four subsets and testing on the fifth. The accuracy of the model is assessed by misclassification error.

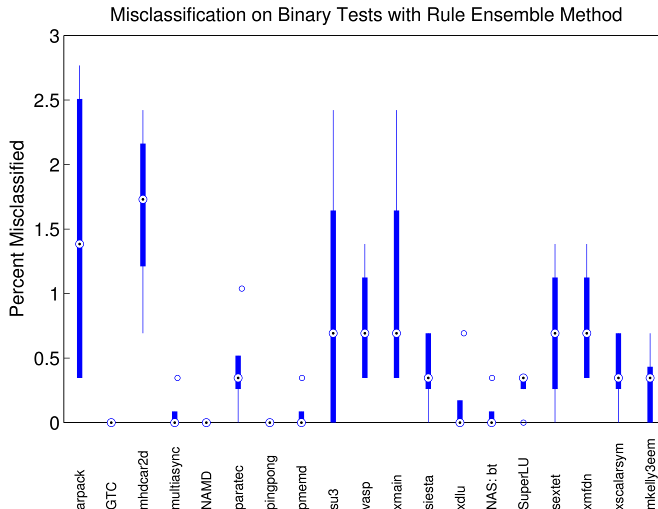

Figure 3: Misclassification error in tests identifying each code from all other possible codes. Each binary test predicts if an observation is or is not part of a given class.

Within each fold of the cross validation we build a binary model for each class that had more observations than folds. Classes that had fewer observations than folds were simply considered as background in the binary models. When binary tests are performed, an overall misclassification rate and false negative error rate are calculated. These rates indicate how biased the model is and if it is overfitting the training data. The false negative rate is particularly important when multi-class problems are framed as binary tests, due to the significant class imbalance that is caused by grouping nearly all the classes into a single group and leaving only one class by itself. The false negative rate indicates if the model overlooks the minority class and simply classifies everything as the majority class.

5. RESULTS

5.1 Identifying Individual Codes

Figure 3 shows the results of performing binary tests for each of the codes. Each binary test predicts if observations are in a given class (i.e. generated by the indicated code), or not in a given class. Immediately we notice that there is a high variance in how easily some codes can be distinguished and that some codes have a more consistent error between folds in the cross-validation. This behavior is evident by the disparity in error rates and the length of the box for each code, respectively. Most notably, GTC, NAMD, Pingpong, and PMEMD have nearly zero error. In contrast, the variance in error between folds is largest for predicting ARPACK and SU3 and the median error is highest for ARPACK and MHDCAR2d.

We also consider the sensitivity of the model to check that the low overall misclassification rate is not an artifact of the significant class imbalance in the dataset. Using OVA

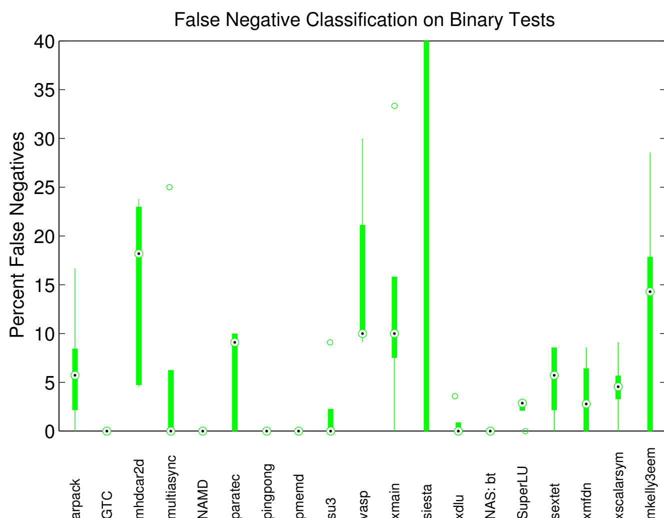

Figure 4: False negative error for binary tests. SIESTA 1-2 observations in a test set, so misclassifying a single observation resulted in 100% error in 3 of the 5 folds.

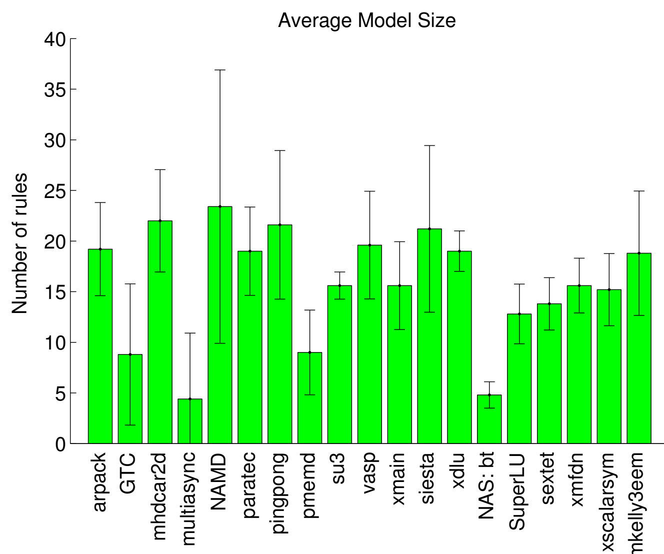

Figure 5: The average number of rules that a model had in each of the cross-validation folds. Fewer rules indicates an easier time identifying that code and allows for better interpretation of of the model and quicker predictions.

classification amplifies class imbalance by trying to identify observations of one code from all the other possible codes. Misclassification error can seem low in such cases if the model blindly classifies all observations, even the very few positive observations, in the negative class. We avoid being misguided by looking specifically at how the model does on identifying the few observations of the positive class. This success can be measured with the percent of false negatives (100 times the number of false negatives divided by the number of positive observations in test set), which is shown in Figure 4.

While significantly larger than the single digit misclassification error, the percent of false negatives in Figure 4 is also relatively low and indicates our models are sensitive. Consider the false negative rate with respect to the class distribution in Figure 1. The classes with fewest observations have only one or two observations in the test set of a single fold. Thus, the false negative rate can appear very large if the test set had a single observation that was misclassified.

5.2 Model Size

In addition to the accuracy of a model, we are interested in its size. The size of the model affects both its computation time and interpretability. A smaller model contains fewer rules and, as a result, takes less time to evaluate. The importance of time cannot be emphasized enough, as lightweight models that are quick to evaluate will be necessary to implement any near-realtime process for classifying codes.

Figure 5 shows the average number of rules that a model had in each of the cross-validation folds. Certain codes require fewer rules and appear easier to identify than others. For example, Multiasync and NAS:bt require relatively few rules while MHDCAR2d, NAMD, Pingpong, and SIESTA require many more rules. In some of the models, Multiasync was identified with a single rule. Such a small model seemed implausible, but Figure 4 shows that these small models were reliably classifying observations. To further verify the results, we looked at the rule and found that the single rule looked at the proportion of time the code spent in `MPI_Test` or `MPI_Testall`. Both of these calls were not used in many of the codes and only Multiasync spent a large portion of the MPI time in this call. The rule ensemble method was able to pick this up in some of the cases and that resulted in remarkably small models. Other codes also used `MPI_Test` and `MPI_Testall`, but few spent a significant amount of time in these calls. Further, not every instance of the other codes used `MPI_Test` and `MPI_Testall`, so the use of these functions was not a distinctive feature of other codes the way it was for Multiasync, for which every instance spent a very large portion of time in these calls.

5.3 Multi-class Classification

We extend the rule ensemble to the multi-class method, as was described in section 3.2. Figure 6 shows that in general the misclassification error is very low, near 5% error. This low error is very good and indicates that codes can reliably be identified from considering only the performance log. Further, the variance between this rate is very small between folds, which indicates the method is reasonably fitting the population and not overfitting the training data in each fold.

5.4 Allowing Observations to be Unclassified

We repeat the tests with the relaxed classification system that allows seemingly anomalous observations to be "unclassified". Relaxing the classification scheme should allow the method to flag observations that it is unsure of for further review, rather than force the method to guess at a classification. The results of allowing the method to not classify observations it is unsure of are compared with the previous

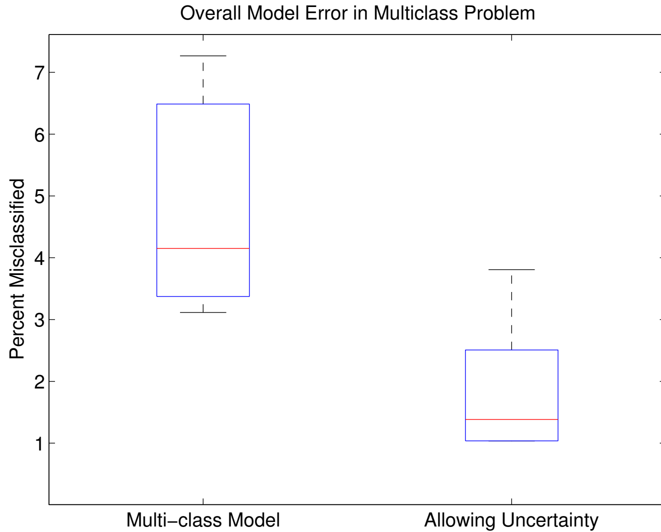

Figure 6: Comparison of the overall misclassification error rate when the model allows uncertainty.

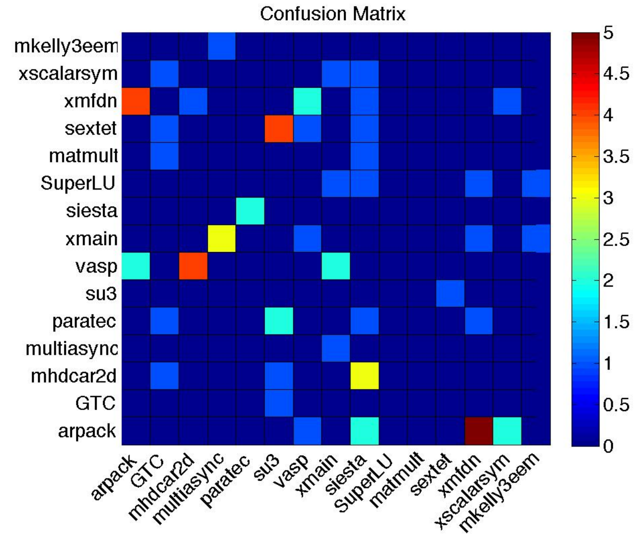

Figure 7: Confusion matrix of misclassifications only. The large majority of applications were correctly classified. The vertical axis is the code that generated a log and the horizontal axis is the predicted label.

results in Figure 6. Less than 5% of observations were left unclassified. Most observations showed affinity to a single class in the binary tests and only a very small percent (less than 3%) of the observations evaluated as positive observations in none or more than one of the binary tests.

5.5 Distribution of Misclassification

Very few observations are misclassified, but it is valuable to consider a confusion matrix to know which and how observations were misclassified. Knowing how observations were misclassified helps ascertain if certain observations were outliers, or if two classes are very similar. The confusion matrix in Figure7 shows only the observations that were misclassified. The vertical axis of the confusion matrix shows the true label of observations and what they were classified as. The vertical axis lists the true label of the code and the horizontal axis indicates the predicted label. The color of each cell indicates how many observations of a certain code are predicted to be in another class with red indicating more observations than blue. There is no strong behavior to regularly misclassify a particular code as another. In general, so few observations of each code were misclassified, that it is difficult to ensure that one code is regularly misclassified as another. This lack of ordered pattern is evident by multiple elements in each row being non-zero.

6. DISCUSSION OF RULES AND ATTRIBUTE IMPORTANCE

Using the importance measure described in section 3.4 we calculate the relative importance of each feature. The distribution of importance across features can be seen in Figure 8. This figure shows the general importance of each feature relative to the other features within each fold of the cross-validation. It also shows that certain features stand out, and these these most important features are described in Table 5.5. The features listed in Table 5.5 had very low importance, consequently little to no predictive capability, and thus were least useful for describing the data. It is interesting that the majority of the least useful features were measures of time. At this preliminary stage it is not yet clear how to normalize for the size of the application run or the number of nodes run on and the list of least predictive features is revealing that this normalization is a problem. Without properly normalizing, the method finds, as we would expect, that the measures of time are very noisy and non-descriptive.

7. RELATED WORK

Many studies have identified that there is distinct behavior in a code's performance logs [10, 16, 14], but have not addressed how unique this behavior is to a code and if similar computations share similar behavior. Vetter and Mueller considered the inherent communication signatures of high performance codes that use MPI [16] and the implications for scalability and efficiency. They looked for characteristic behavior by examining the point-to-point messages, collec-

Importance	Attribute
2.4346	MPI time spent in `MPI_Waitall`
1.8101	% run time spent in MPI calls
1.5152	% of MPI spent in collective calls
1.2904	number of `MPI_Reduce` calls
1.2252	(average gflops/sec) * (# nodes)
1.1998	% of MPI time spent in `MPI_Reduce`
1.0738	% of MPI time spent in `MPI_Wait`
1.0028	% of MPI time spent in `MPI_Sendrecv`
1.0007	% of MPI time spent in `MPI_Test`

Table 6: Attributes with the largest relative importance are the most useful for building an accurate model.

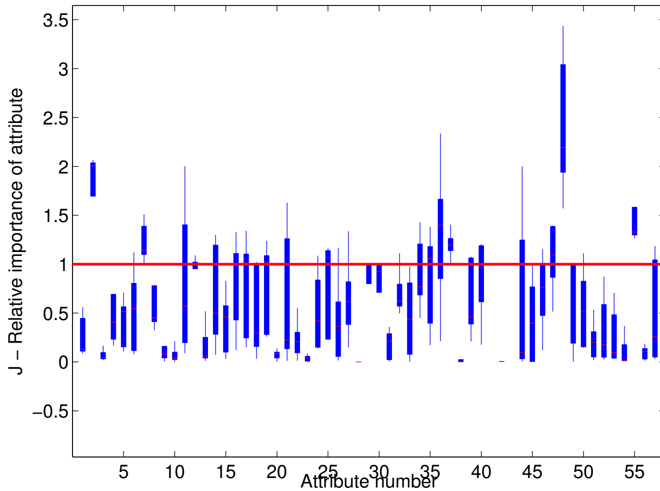

Figure 8: We get a measure of how important an attribute is for identifying each code and then aggregate these into a general measure of overall importance. The red line indicates 1 and very few attributes have a general importance larger than this. The attributes with mean larger than 1 are listed in table 5.5.

tive messages, and performance counters. This study was done for 5 example applications. Kamil et. al [10] and Shalf et. al [14] performed similar studies of MPI communication for suites of six codes and eight codes respectively. Kamil et. al and Shalf et. al used codes from a range of scientific applications and computational families, e.g. matrix computations, particle in cell codes, and finite difference on a lattice, to examine what resources these computations needed and what future architectures would have to provide.

In particular, few studies have utilized supervised learning algorithms to understand performance logs of high performance codes. A study by Whalen et. al [17] successfully used graph isomorphisms and hypothesis testing to characterize codes that use MPI and identify such codes based on their IPM performance logs. Whalen et. al used the hashed entries of point-to-point communication from the IPM logs to generate data points. Each run of a code was then a set of points and these points were used to predict which code was used to generate a given log.

A study by Peisert [12] suggested the utility of supervised machine learning for performance logs. This preliminary study used a variety of methods on a limited set of logs and achieved good classification results, but indicated that a more extensive study was needed.

Importance	Attribute
0.0011	time in `MPI_Gatherv`
0.0039	time in `MPI_Test`
0.0217	time in `MPI_Rsend`
0.0370	time in `MPI_Bcast`
0.0682	range in time each node spends in user calls
0.0683	time in `MPI_Allgatherv`
0.0730	time in `MPI_Barrier`
0.0921	wall clock time spent in collective calls

Table 7: These attributes had the least importance and consequently little predictive capability.

Our work continues in the nature of Whalen et. al and Peisert, but uses a larger set of observations and codes. We also focus on data of a much higher granularity. Instead of detailed information about each communication, we use high level performance statistics that are more easily captured and less expensive to collect and process. We also use rule based supervised learning approaches that have been highly successful in a variety of other domains rather than graph theoretic approaches.

8. CONCLUSIONS

We have showed that it is possible to fingerprint HPC codes by simply looking at their performance log. With very high accuracy, our method tells which code generated a given performance trace. The attributes used for this prediction were from fairly non-intrusive IPM logs and could be readily available with a very low runtime overhead cost.

The rule ensemble method and the extensions that we utilized were able to automatically identify blatant characteristics in the dataset, such as Multiasync spending a majority of time in `MPI_Test`. The method was also able to reliably classify applications that did not have as blatantly unique of a signature. Some of the applications varied between instances, as can be expected of large applications that have more capability than is used in every execution of the code, and the method was still able to classify those codes. The flexibility of the ensemble method was able to identify these codes even though in some instances they used certain calls and in other instances, when another portion of the code was called, other calls were used.

Using this particular machine learning approach was useful because it gave us insight into which features were useful for classifying the performance logs. The most important features were MPI call specific and focused on how the code communicated with the MPI library. Intuitively this seems reasonable, but it implies that the method may be classifying codes based on how they were programmed rather than more general computational traits. More work must be done to generalize the feature vector. Noting that the features which were deemed least important were mostly measures of time, reinforced our assumption that logs must be appropriately normalized for the number of nodes they ran on, time they ran, etc. before all the information can be utilized.

9. FUTURE WORK

We would like to further explore how not to force an anomalous observation to be classified into one of the previously seen classes. It is not clear how to process unclassified observations and refine the classification to ensure that the separated observations are actually anomalous.

More work needs to be done with feature selection and understanding the granularity of performance data that is needed. We chose to use attributes that describe the code's behavior on a very high level. This decision was made with the intention of minimizing the necessary disruption of profiling a

code if these methods were included in a monitoring system. There may be other features that more succinctly define the behavior of a code and that could also be collected at a low overhead cost.

We would like to try an alternative transition from the binary to the multi-class model by using multi-class decision trees to fit rules rather than binary trees for each class. It is possible that rules from multi-class trees will be better formed. Building a single set of rules will also significantly decrease training and testing time, as well as allow for a more cohesive rule and attribute ranking system.

Finally, we would like to look at classifying codes into computational classes rather than specific code. Such classification would be more broad and would be applicable in the area of optimization and auto-tuning. Automatically identifying computational families could enable administrators to suggest optimizations and tools to scientists that have been designed for specific computational patterns.

10. ACKNOWLEDGMENTS

We would like to thank Sean Peisert, Scott Campbell, and David Skinner for very valuable conversations. This research was supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC02-05CH1123.

11. REFERENCES

[1] Correspondence with ipm developers.

[2] Ipm: Integrated performance monitoring.

[3] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[4] O. DeMasi, J. Meza, and D. Bailey. Dimension reduction using rule ensemble machine learning methods: A numerical study of three ensemble methods. 2011. `http://crd.lbl.gov/~dhbailey/dhbpapers/Ensemble_TechReport.pdf`.

[5] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer Series in Statistics, 2001.

[6] J. Friedman and B. Popescu. Importance sampled learning ensembles. Technical report, Department of Statistics Stanford University, 2003. `http://www-stat.stanford.edu/~jhf/ftp/isle.pdf`.

[7] J. Friedman and B. Popescu. Gradient directed regularization. Technical report, Department of Statistics Stanford University, 2004. `http://www-stat.stanford.edu/~jhf/ftp/pathlite.pdf`.

[8] J. Friedman and B. Popescu. Predictive learning via rule ensembles. *Annals of Applied Statistics*, 2(3):916–54, 2008.

[9] E. Hale, W. Yin, and Y. Zhang. Fixed-point continuation (fpc) an algorithm for large-scale image and data processing applications of l1-minimization. `http://www.caam.rice.edu/~optimization/L1/fpc/`

[10] S. Kamil, J. Shalf, L. Oliker, and D. Skinner. Understanding ultra-scale application communication requirements. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 178–187. IEEE, 2005.

[11] K. Keutzer and T. Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal*, 13:4, 2010.

[12] S. Peisert. Fingerprinting communication and computation on hpc machines. 2010. `http://escholarship.org/uc/item/7121p0xd.pdf`.

[13] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *The Journal of Machine Learning Research*, 5:101–141, 2004.

[14] J. Shalf, S. Kamil, L. Oliker, and D. Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybrid interconnect. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 17. IEEE Computer Society, 2005.

[15] D. Skinner. Performance monitoring of parallel scientific applications. *Tech Report [LBNL-5503], Lawrence Berkeley National Laboratory*, 2005.

[16] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, 2003.

[17] S. Whalen, S. Engle, and M. Peisert, S.and Bishop. Network-theoretic classification of parallel computation patterns. *International Journal of High Performance Computing Applications*, 2012.