Title

Multiclass Classification of Distributed Memory Parallel Computations

Journal

Pattern Recognition Letters (PRL), 34(3)

Authors

Whalen, Sean
Peisert, Sean
Bishop, Matt

Publication Date

2012-11-05

Peer reviewed

Multiclass Classification of Distributed Memory Parallel Computations

Sean Whalen[a,*], Sean Peisert[b,c], Matt Bishop[c]

[a] *Computer Science Department, Columbia University, New York NY 10027*
[b] *Lawrence Berkeley National Laboratory, Berkeley CA 94720*
[c] *Computer Science Department, University of California at Davis, Davis CA 95616*

Abstract

High Performance Computing (HPC) is a field concerned with solving large-scale problems in science and engineering. However, the computational infrastructure of HPC systems can also be misused as demonstrated by the recent commoditization of cloud computing resources on the black market. As a first step towards addressing this, we introduce a machine learning approach for classifying distributed parallel computations based on communication patterns between compute nodes. We first provide relevant background on message passing and computational equivalence classes called *dwarfs* and describe our exploratory data analysis using Self Organizing Maps. We then present our classification results across 29 scientific codes using Bayesian networks and compare their performance against Random Forest classifiers. These models, trained with hundreds of gigabytes of communication logs collected at Lawrence Berkeley National Laboratory, perform well without any a priori information and address several shortcomings of previous approaches.

Keywords: Multiclass classification, Bayesian networks, Random forests, Self-organizing maps, High performance computing, Communication patterns

*Corresponding author. Phone: 1 212 939 7078.
Email addresses: `swhalen@cs.columbia.edu` (Sean Whalen), `sppeisert@lbl.gov` (Sean Peisert), `bishop@cs.ucdavis.edu` (Matt Bishop)

1. Introduction

U.S. Government laboratories operate 4 of the 10 most powerful known High Performance Computing (HPC) systems in the world according the TOP500 supercomputer ranking website. The massive computational power of these resources makes them potential targets for attackers as evidenced by the growing black market for encryption and password cracking services. For example, various websites offer to crack WPA-PSK or ZIP passwords in 20 minutes for $17 using a cloud computing infrastructure. Knowing what codes and algorithms are running on these systems is thus extremely important, yet this poses a deceptively difficult problem.

There are also more benign reasons for fingerprinting HPC programs. Compute time on such systems is at a premium and users are often granted access for a specific purpose. Running unauthorized codes, or running classified codes on unclassified systems, may be against policy. Determining what is running may not be a trivial task: codes are often compiled with default compiler names such as `a.out` and can simply be renamed to appear as another binary. There are also performance reasons. Certain algorithms may be inefficient on a particular platform and have tuned implementations available. Identifying the underlying algorithm could enable automatic algorithm replacement [1, 2] and save researchers valuable compute time.

Though we are not primarily concerned with security applications here, our work is motivated by *anomaly detection*: given some unknown communication pattern, we compute the likelihood it was generated by each of a set of models constructed from authorized programs. The pattern is labeled anomalous if its posterior likelihood is not above a certain threshold for at least one of these models. Fingerprinting these communications is the more general problem of structural pattern recognition, where an unknown distributed computation re-

veals itself indirectly via messages exchanged by a network of compute nodes. The topology of the network and properties of these messages are typically well structured.

Identifying the computation underlying some pattern of distributed communication is thus a type of latent class analysis where a "hidden" algorithm must be identified only from observable information flows on the network. In this paper we continue previous work based on graph theory, network theory, and hypothesis testing [3] by using machine learning to identify the unknown algorithm most likely underlying these observed patterns of communication. We first review message passing and computational equivalence classes called *dwarfs* and discuss how dynamic communications are captured from running applications. We then present our exploratory data analysis using a non-linear generalization of Principal Component Analysis called Self-Organizing Maps. Finally, we present the classification results for 29 scientific codes from Lawrence Berkeley National Laboratory using Naïve Bayes, Tree-Augmented Naïve Bayes, and Random Forest classifiers. For many applications of this work, identification of the computational dwarf may be of more interest than identifying individual codes. This should lead to even higher classification performance and is discussed in Section 6.

2. Related Work

Communication patterns have previously been used to study the performance of distributed memory applications, though the application of machine learning to the area has likely been limited by the difficulty of data collection. Fürlinger et al. [4] provide a general introduction to communication logging and discuss several concepts related to this work including visualization of adjacency matrices and examining the distribution of aggregate communications. Shalf et

al. [5] perform similar analysis to evaluate the communication requirements of parallel programs for improving processor interconnect designs. The adjacency matrices of several parallel benchmark applications, augmented by number of messages and message size, are presented by Riesen [6].

Ma et al. [7] introduce a communication correlation coefficient to characterize the similarity of parallel programs using several metrics. The first compares the average transmission rate, message size, and unique neighbor count for each rank, while the second computes the maximum common subgraph. Their evaluation was limited to 4 parallel benchmark applications. Similar in scope, Florez et al. [8] trained neural network and Hidden Markov Model classifiers on communications and system calls to flag anomalous behavior in 2 parallel programs.

We present elsewhere classification approaches using graph theory, network theory, and statistical hypothesis testing [3] operating on attributed relational graphs where nodes are ranks and edges are MPI calls. These methods are more computationally efficient but less accurate than the machine learning approaches presented here.

Our work examines applications using the Message Passing Interface (MPI) standard for distributed memory programming. Other parallel programming standards such as OpenMP are based on a shared, as opposed to distributed, memory model. In an effort to increase the portability of parallel software, recent work uses compiler techniques to translate OpenMP into MPI source code [9, 10], and our approach should apply when such techniques are used. While we focus on latent analysis using only runtime communications, source code translation has also been used to replace inefficient computations [1, 2]. The classification of distributed computation patterns thus has strong ties to compilers, static analysis, and code optimization.

3. High Performance Computing

3.1. Message Passing Interface

Message Passing Interface (MPI) is a communications protocol standard used by many parallel programs to exchange data using a distributed memory model. There are several implementations such as OpenMPI and MPICH, each based on the idea of logical processors with unique labels called *ranks* placed in groups called *communicators*. MPI programs have an initialization phase where each processor joins a communicator and is assigned a rank, and a finalization phase to gracefully terminate after computation.

The *Integrated Performance Monitoring* (IPM) library [11] provides low overhead performance and resource profiling for parallel programs. It logs features of MPI calls such as the call name, the source and destination rank, the number of bytes sent, and aggregate performance counters such as the number of integer and floating point operations. The library is enabled at compile time and uses library interposition to intercept MPI calls at runtime.

Consider the following abbreviated IPM log entry:

<hent call="MPI_Isend" bytes="599136" orank="1" count="26" />

These entries become rows in a two dimensional feature matrix where rows are individual calls and columns are call features. Call names are mapped to unique integers so the contents of the feature matrix are purely numerical. The above entry then becomes:

$$\begin{pmatrix} \text{int(MPI\_Isend)} & 599136 & 1 & 26 \end{pmatrix}$$

The result is a matrix of features for each run of a parallel program. By varying datasets, parameters, the number of compute nodes, and other factors, we obtain multiple matrices for each program. The task at hand, then, is to differentiate

patterns of parallel computation while recognizing how the same program may express multiple patterns under different conditions.

3.2. Computational Dwarfs

A computational dwarf is "a pattern of communication and computation common across a set of applications" [12]. Each dwarf is an equivalence class of computation independent of the programming language or numerical methods used for a particular implementation. The common use of shared libraries such as BLAS and LAPACK provides some evidence of these equivalence classes, though dwarfs imply a level of algorithmic equivalence beyond code reuse.

Colella et al. identified seven dwarfs in HPC applications [13]: dense linear algebra, sparse linear algebra, spectral methods, n-body methods, structured grids, unstructured grids, and monte carlo methods. Asanovi et al. asked if these seven also captured patterns from areas outside of HPC [12]. They found six additional dwarfs were needed to capture the distinct patterns of computation outside HPC including combinational logic, graph traversal, dynamic programming, backtrack and branch/bound, graphical models, and finite state machines.

Distributed memory parallel programs, then, will fall into one or more of these 13 dwarf classes. If the variance of the expressed patterns is bounded, identification of the dwarf class should be possible solely from observed communications.

4. Exploratory Data Analysis

4.1. Adjacency Matrices

Consider a three node communicator where rank 0 sends messages to ranks 1 and 2, rank 1 sends a message to rank 2, and ranks 1 and 2 send messages back

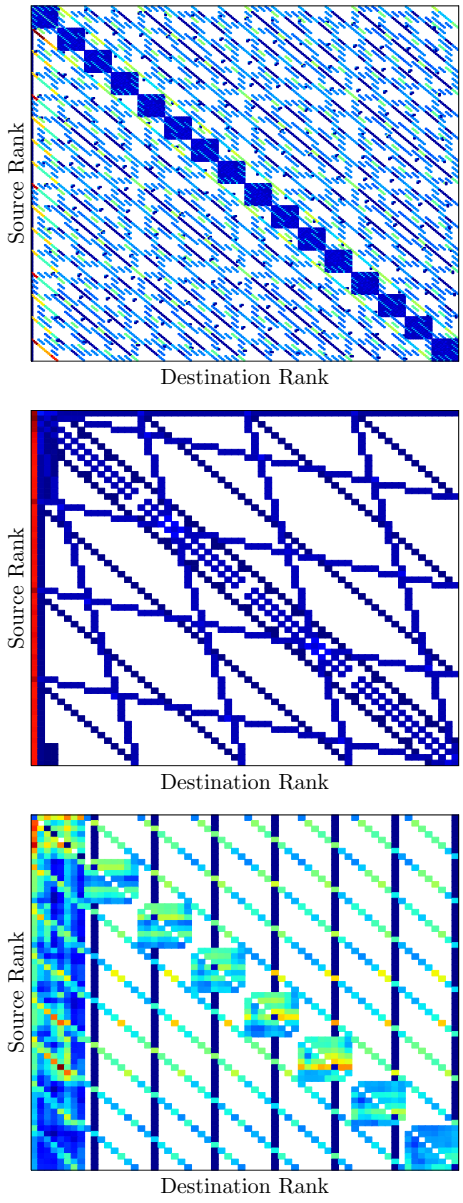


Figure 1: Adjacency matrices for individual runs of astrophysics benchmark MADBENCH (256 nodes), atmospheric dynamics simulator FVCAM (64 nodes), and linear equation solver SUPERLU (64 nodes). The number of bytes sent between ranks is linearly mapped from dark blue (lowest) to red (highest), with white indicating an absence of communication.

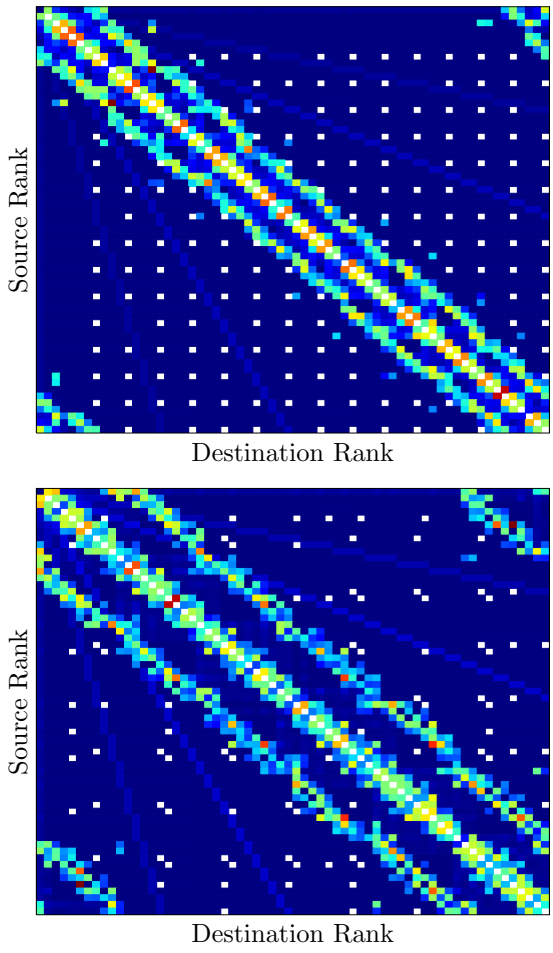


Figure 2: Data dependent topology demonstrated by molecular dynamics simulator NAMD under different molecular arrangements. The number of bytes sent between ranks is linearly mapped from dark blue (lowest) to red (highest), with white indicating an absence of communication.

to 0. These messages have the following adjacency matrix representation:

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Adjacency matrices are commonly visualized as a grid where the axes are rank numbers and filled pixels denote ranks that exchanged one or more messages. Different communication features such as the number of messages exchanged or their total size can be stored in the matrix and color-mapped to provide additional insight. Such visualizations are commonly used to examine communication patterns and have been offered as evidence for the existence of computational dwarfs. The adjacency matrices for single runs of three different parallel programs are shown in Figure 1.

Communication patterns are strongly tied to distributed memory access within a parallel program. To see this, examine the diagonal of Figure 1's center panel and note the communication between a rank and its immediate neighbors. Such a pattern is generated by finite difference equations and is found across many HPC applications. Another type of equation will have a different visual signature unless its pattern of distributed memory access is similar.

The structure seen in Figure 1 is typical of MPI applications and suggests that classification is possible. By the same argument, however, distinguishing applications within the same dwarf class may be difficult due to their topological similarity. Complicating matters, the same program may alter its communications given different parameter values, datasets, or communicator sizes (see Figure 2). As a result, we cannot simply compare adjacency matrices to classify the underlying computation.

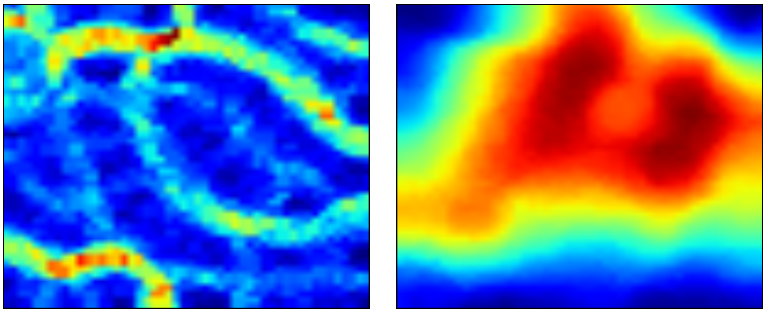
4.2. Self-Organizing Maps

To explore relationships between features we perform unsupervised clustering using a type of neural net called a *Self-Organizing Map* (SOM) [14]. In contrast to more common clustering algorithms such as k-means which require specifying the number of clusters (k) a priori, SOMs infer the number of clusters directly from the data. These clusters are visualized by projecting high dimensional inputs onto a two dimensional grid while simultaneously preserving the topological properties of the input space. Thus, they provide both clustering and dimensionality reduction and are a non-linear generalization of Principal Component Analysis [15].

The two dimensional grid consists of randomly valued n-dimensional vectors of weights where n is the dimension of the input space. The weights are trained iteratively using a competitive learning algorithm: A random input vector is selected, the closest vector on the grid (the "winner") is computed, and both the winner and its neighbors adjust their values towards the input vector. Both the number of neighbors selected and the amount they change decrease each iteration until training converges.

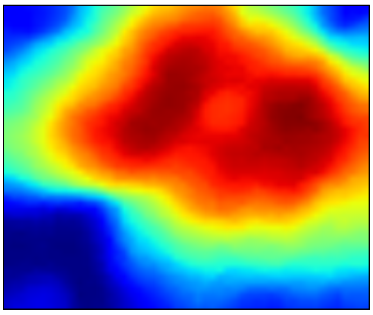
If the input space is three dimensional then the weights of each trained vector in the grid can be visualized as the red, green, and blue components of a pixel. For input spaces beyond three dimensions there is no such convenient mapping. Instead, the *U-matrix* [14] stores the average L2 distance between each trained vector and its nearest neighbors. When the U-matrix is viewed as a color mapped grid, clusters emerge as high contrast curves.

SOMs may only provide a qualitative view of clustering depending on data complexity, properties of the grid, and training parameters. Such is the case for the U-matrix of CACTUS shown in Figure 3a where small values are mapped to blue and large values to red. While clustering structure is visible, the exact

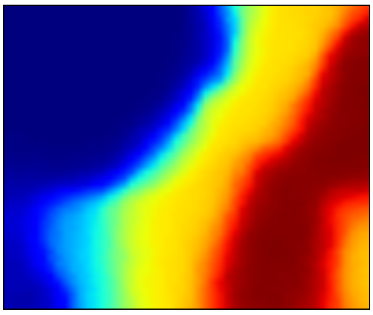


(a) U-matrix

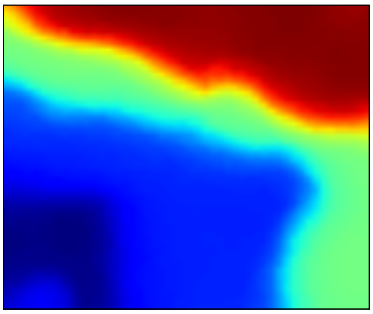
(b) Source Rank



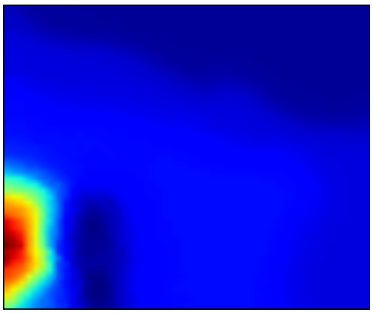
(c) Destination Rank



(d) MPI Call



(e) Bytes Sent



(f) Repeat Count

Figure 3: U-matrix and feature layers of a self organizing map trained on data from general relativity simulator CACTUS. Clusters emerge as high contrast curves in the U-matrix; feature layers show correlations due to the topology-preserving projection of the SOM. Small values are mapped to blue and large values to red.

number of clusters is subject to interpretation.

In such instances, viewing the correlations between features can be more informative. This is accomplished by viewing individual "layers" of the trained grid corresponding to particular input dimensions. For example, the fourth component of each 5-dimensional input vector is the number of bytes sent between two ranks for a particular MPI call. Viewing only the fourth dimension of the trained vectors demonstrates how the inputs cluster with respect to the number of bytes sent.

These per-feature layers are shown in panels (b) through (f) of Figure 3 (again with small values mapped to blue and large values to red). The southwest corner of the "repeat count" layer reveals a red region corresponding to frequently repeated messages. Since the 2-dimensional projection preserves the topology of the input space, the low intensity of the same region in the "bytes sent" layer tells us that frequently repeated messages are also very small. In concert with the U-matrix, these feature layers make SOMs a powerful tool for discovering the structure and relationships hidden away in high dimensional data.

Models that account for the feature correlations observed above should perform better than those that assume independence. Along with the computational tradeoffs necessitated by the size of our dataset, this motivates our chosen model class introduced in the next section.

5. Bayesian Networks

In previous work [3] we found the distribution of MPI calls relative to each source rank were sufficient for accurate classification using goodness-of-fit tests. This approach has two potential downsides. First, the chosen test generates conservative p-values when used with discrete distributions. Second and more

significantly, without incorporating correlations between features it becomes less likely that codes will remain distinguishable as the number of codes increases.

This section reviews models called Bayesian networks that better approximate the joint communication feature distribution. We later evaluate these models and compare their classification performance to independent feature models as well as ensembles of decision trees called Random Forests [16]. The latter are state-of-the-art in many classification tasks and so serve as a useful reference point to compare against Bayesian networks in this setting.

Bayesian networks are widely used graphical models that allow efficient factoring of joint probability distributions by using directed acyclic graphs to represent conditional dependencies (the edges) between random variables or *features* (the nodes) [17, 18]. The joint probability of random variables $X_1, \ldots, X_v$ can be factored using the Markov property that limits the conditional dependencies of variable X_i to its parent nodes Π_{X_i} [19]:

$$P(X_1, \ldots, X_v) = \prod_{i=1}^{v} P(X_i | \Pi_{X_i})$$

The simplest Bayesian network, the Naïve Bayes [20] model, assumes all nodes are conditionally independent given a common parent node representing the class label. It is sometimes called an *independent feature model* as a result. In contrast, the full joint distribution is represented as a completely connected graph. Between these extremes one can learn only the dependencies deemed relevant by some statistical test, as well as estimate conditional probabilities using Maximum Likelihood Estimation or smoothing techniques such as Dirichlet priors [18].

Structural learning algorithms for Bayesian networks are broadly categorized into traditional heuristic searches optimizing some scoring function and specialized constraint-based searches that restrict the space of dependence relations.

Examples of the latter include Chow-Liu (CL) trees [21] and Tree-Augmented Naïve Bayes (TAN) networks [22]. These are optimal tree-structured approximations of the joint distribution and can be learned in $\mathcal{O}(n^2)$ time. Both CL and TAN perform similarly in practice [22] though TAN generally performs better for classification tasks. We briefly review the TAN learning algorithm here due to its superior performance detailed in the next section.

Naïve Bayes assumes conditional independence between nodes given a parent node representing the class label. For example, 5 features are represented by a graph with 6 nodes (1 per feature plus 1 for the class label) and 5 edges (1 from the class node to each feature node). The only correlations captured are between the class label and individual features; no correlations between features are accounted for though they often exist in practice.

Edges may be added to account for these correlations and improve model accuracy, but we often want to minimize the number of added edges for efficiency. The TAN algorithm finds the most informative edges to add given some training data by first computing the conditional mutual information [23] between each pair of features X and Y and the class label Z:

$$I(X;Y|Z) = \sum_{x,y,z} P(x,y,z) \log_2 \frac{P(x,y|z)}{P(x|z)P(y|z)}$$

It then constructs a complete undirected graph G with the conditional mutual information labeling the edge between X and Y. The maximum weight spanning tree is computed to retain only the most informative edges from the space of possible tree topologies. Edges not in the tree are dropped from G. A root is arbitrarily selected and undirected edges are given an outward direction from this root to convert G into a Bayesian network. Finally, a node for the class label is added as well as outward edges from this new label node to each feature node. Thus TAN "augments" the Naïve Bayes dependency graph by overlaying

an optimal set of tree-structured edges.

We now have a Bayesian network whose structure incorporates a set of additional dependencies that more accurately represents the joint distribution of the data with a limited increase in the number of parameters. TAN modifies the Chow-Liu algorithm by using conditional instead of unconditional mutual information, as well as adding a node and edges for the class label.

6. Evaluation

All models were constructed and evaluated using the WEKA machine learning toolkit v3.7.5 [24] trained on IPM logs. As described in Section 3.1, logs are converted into feature matrices with each row corresponding to an MPI call made between a source and destination machine during the execution of a particular code. Broadcast messages are supported by IPM via use of a negative destination rank number.

Hundreds of gigabytes of logs proved problematic for the non-distributed nature of WEKA, so some preprocessing was required. In addition, the data has class imbalances due to the disparate number of logs available for each code. To address these issues we use WEKA's supervised instance resampling filter to both up- and down-sample classes, creating a feature matrix containing 5% of the original data and with a uniform class distribution.

Many structural learning algorithms including TAN require discrete features. While our features are already discrete, the message size feature can take on millions of values. Applying WEKA's supervised discretization filter based on Fayyad et al. [25] greatly reduces the number of unique values and makes computing conditional probabilities feasible.

Having balanced and discretized data, we evaluate the algorithms by generating a *confusion matrix* using WEKA. The confusion matrix is a table that

shows the actual versus predicted labels for a supervised learning algorithm. Evaluating the performance of a classifier using the confusion matrix is often straightforward, but is more involved if there are more than 2 classes or if the classes are unevenly distributed. The latter scenario is handled by the re-sampling method described in the previous section as well as the use of the evaluation method outlined below. To handle more than two classes, WEKA converts the multi-class confusion matrix into a set of binary matrices using the *one-versus-rest* approach [26].

The 2-by-2 matrix for some program α shows the number of instances correctly and incorrectly classified as α versus all programs except α. A *true positive* (tp) occurs when an MPI call from program α is classified as program "not α"; a *false positive* (fp) occurs when a call from "not α" is misclassified as α. Similarly, a *true negative* (tn) occurs when a call from "not α" correctly fails to be classified as α, and a *false negative* (fn) occurs when a call from α is incorrectly classified as "not α". Two common statistics for evaluating classifiers using these quantities are the precision P and recall R:

$$P = \frac{tp}{tp + fp}$$
$$R = \frac{tp}{tp + fn}$$

Their harmonic mean is called the F_1 score and is used to summarize performance with a single number:

$$F_1 = 2\frac{PR}{P + R}$$

The F_1 scores for Naïve Bayes, Tree-Augmented Naïve Bayes, and Random Forest classifiers are shown in Table 1. These are 80.7%, 88.1%, and 88.3%, respectively. Each score is obtained from 5-fold stratified cross validation [26]. This

process builds multiple classifiers using different training and test sets (folds) and averages their scores to increase confidence that the measured performance will generalize to unseen data. It is clear from the cross-validated F_1 scores that feature correlations are important for discriminating between classes, as evidenced by the comparatively poor performance of Naïve Bayes.

A total of 1681 logs for 29 scientific applications were collected for Lawrence Berkeley National Laboratory by the National Energy Research Scientific Computing Center. Though a single dwarf class for each node is listed for compactness, many codes belong to more than one class. Multiple logs exist for each code with varying ranks, parameters, architectures, and datasets when possible. Several simpler codes were logged by us; codes requiring significant domain knowledge or private datasets were logged from willing specialists on production systems. As a result, the inputs and parameters for some codes were not under our control. However, this dataset is several orders of magnitude larger than related efforts and contains a representative sample of the dwarfs found in scientific computing.

Examining the confusion matrix (ommitted for size) reveals that most classification errors are due to codes within the same dwarf class, or multi-dwarf codes that make use of the same library such as FFTW [27], SUPERLU [28], or SCALAPACK [29]. For example, both the PARATEC [30] and VASP [31] are materials science codes. PARATEC is the hardest code to classify, having a recall of 63.1%. We can attribute 11% of its false negatives to VASP and another 11% to linear algebra operations in 2 other programs: 22% of its false negatives are due to only 3 other codes. Thus, if one wanted to infer the computational class of some unknown program rather than the exact code, classifcation performance would be substantially higher. This coarser-grained information would still be very useful for an anomaly detection system.

Table 1: Multi-class performance of Naïve Bayes, Tree-Augmented Naïve Bayes, and Random Forest classifiers. A higher F_1 score indicates better classification performance. Each score is averaged over a 5-fold stratified cross-validation using a class-balanced sampling from 5% of the original dataset. The increased complexity of the tree-structured network provides significant improvement over the independence assumption of Naïve Bayes.

			Ranks		F_1 Score		
Code	Area / Library	Primary Dwarf	Min	Max	NB	TAN	RF
CACTUS	Astrophysics	Structured Grids	64	256	0.928	0.932	0.944
FVCAM	Atmospheric Dynamics	Structured Grids	64	64	0.872	0.923	0.957
GTC	Magnetic Fusion	Unstructured Grids	64	256	0.964	0.964	0.976
GTS	Magnetic Fusion	Unstructured Grids	64	2048	0.884	0.924	0.943
GRAPH500	Graph Theory	Graph Traversal	256	484	0.845	0.95	0.944
HYPERCLAW	Gas Dynamics	Structured Grids	256	256	0.856	0.927	0.918
IJ	HYPRE	Sparse Linear Algebra	64	4096	0.749	0.812	0.855
IMPACT-T	Accelerator Physics	Structured Grids	256	1024	0.898	0.927	0.951
LBMHD	Hydrodynamics	Structured Grids	64	256	0.875	0.957	0.962
MADBENCH	Astrophysics	Dense Linear Algebra	256	256	0.888	0.944	0.962
MAESTRO	Hydrodynamics	Structured Grids	8	2048	0.777	0.861	0.866
MFDN	Nuclear Physics	n-Body Methods	1	6441	0.725	0.87	0.871
MHDCAR	Plasma Physics	Structured Grids	1	2048	0.753	0.829	0.83
MILC	Lattice Gauge Theory	Structured Grids	64	1024	0.859	0.922	0.94
NAMD	Molecular Dynamics	n-Body Methods	32	128	0.928	0.932	0.944
PARATEC	Materials Science	Spectral Methods	1	256	0.62	0.723	0.728
PDDRIVE	SUPERLU	Sparse Linear Algebra	64	64	0.697	0.805	0.819
PDGEMM	SCALAPACK	Dense Linear Algebra	64	64	0.906	0.945	0.963
PDSYEV	SCALAPACK	Dense Linear Algebra	64	64	0.754	0.888	0.942
PF_2	Atomic Physics	Dense Linear Algebra	1	4096	0.67	0.799	0.811
PMEMD	Molecular Dynamics	n-Body Methods	64	256	0.721	0.801	0.773
PSI-TET	Fluid Dynamics	Unstructured Grids	2	128	0.813	0.864	0.852
PSTG3R	Atomic Physics	Dense Linear Algebra	48	960	0.791	0.901	0.837
TGYRO	Magnetic Fusion	Unstructured Grids	16	16384	0.792	0.909	0.917
TRISTAN-MP	Plasma Physics	Structured Grids	16	4096	0.932	0.946	0.941
VASP	Materials Science	Spectral Methods	1	256	0.702	0.76	0.783
VORPAL	Plasma Physics	Unstructured Grids	1	3072	0.814	0.878	0.896
WAVE	Hydrodynamics	Structured Grids	64	1024	0.632	0.816	0.756
XDLU	SUPERLU	Sparse Linear Algebra	64	64	0.724	0.821	0.802
Weighted Average					0.807	0.881	0.883

Classifier error can be understood as a tradeoff between bias and variance. A high bias, low variance algorithm produces consistent classification across different datasets (low variance) at the cost of having consistently worse labeling (high bias) due to more general decision boundaries. In contrast, a low bias algorithm may perform well on certain datasets and worse on others: it finds better decision boundaries at the risk of overfitting the training data. In general, bias and variance cannot be simultaneously optimized and thus the choice of learning algorithm depends on the complexity and size of the data as well as the requirements of the task.

To better understand the performance of our classifiers, we measure the bias-variance tradeoff using the technique of Kohavi and Wolpert [32] implemented in WEKA. Specifically, the data is first divided in two. The latter half of the data is held constant for testing while 25 training sets are generated from the first half by resampling without replacement. The bias and variance are estimated by evaluating classifiers trained on these separate training sets against the same test set. These estimates are shown in Table 2. As one would expect from the F_1 scores, Naïve Bayes has the highest bias and the lowest variance due to its simple decision boundaries. Tree-Augmented Naïve Bayes trades a large decrease in bias for a slight increase in variance, while Random Forests slightly improve this bias for a larger increase in variance. This too is consistent with the F_1 scores: when the two classifiers differ, Random Forests tend to perform much better or much worse for particular classes.

These results are quite positive and are comparable with our previous work using graph theory, network theory, and hypothesis testing. However, machine learning is computationally more efficient than our graph- and network-theoretic approaches, resolves several shortcomings with the hypothesis testing approach, and achieves these results with a much larger and diverse dataset. We expect

Table 2: Bias-variance decomposition for Naïve Bayes, Tree-Augmented Naïve Bayes, and Random Forest classifiers. Numbers were obtained using the approach of Kohavi and Wolpert [32] implemented in WEKA. The data is first divided in two and the latter half held constant for testing. The first half is resampled without replacement to generate 25 training sets. Bias and variance is estimated by evaluating classifiers trained on these separate sets against the same test set.

Classifier	Bias^2	Variance
Naïve Bayes	0.1944	0.0103
Tree-Augmented Naïve Bayes	0.1233	0.0118
Random Forest	0.0935	0.0476

more difficulty when distinguishing codes sharing a dwarf class such as PARATEC and VASP, and these bring the average F_1 score down considerably. Training with larger subsets of the original data, using search-based structural learning algorithms, or constructing Bayesian multi-nets [22] could increase performance in these circumstances. We leave these investigations for future work.

7. Conclusion

This work applies methods from machine learning to identify the latent class of a parallel computation from the observable information passed between nodes in a computational network: given logs of MPI messages from an unknown program, our job is to infer the program most likely to have generated those logs. Our motivation is the detection of anomalous behavior on HPC systems, though additional applications such as performance profiling are possible.

As initially postulated by work on computational dwarfs [12, 13], communication patterns tend to be highly structured and reflect the distributed memory access patterns of the underlying algorithm. When dealing with algorithm *implementations*, however, many other factors affect the communication patterns of theoretical algorithms. Different implementations of the same computation, shared libraries, compiler optimizations, architecture differences, software flaws, debug flags, and numerous MPI implementations all make this task more dif-

ficult. Further, some parallel programs have data-dependent communication topologies (see Figure 2), varying both slightly and greatly as with multi-use ("swiss-army") libraries or interpreters such as Matlab.

We trained Naïve Bayes, Tree-Augmented Naïve Bayes, and Random Forest classifiers on hundreds of gigabytes of runtime communication logs covering 29 distributed memory scientific codes from Lawrence Berkeley National Lab. This required both up- and down-sampling the data to correct for class imbalance and allow processing by a single computer using WEKA, as well as discretization of numeric features to feasibly compute conditional probability tables. TAN and Random Forest classifiers achieved 88.1% and 88.3% F_1 scores averaged over 5-fold stratified cross-validation, while using only 5% of the original dataset. These results are encouraging and improve upon several weaknesses in our previous work [3] testing the goodness-of-fit of call distributions as well as the distribution of over-represented subgraphs called *motifs* [33, 34]. In addition, classification of computational *dwarfs* instead of individual *codes* should have substantially higher performance, as most errors in the confusion matrix are due to codes within the same dwarf class. In fact, the latter may be more relevant than focusing on individual codes for many potential applications of this work. In either case, we believe these results demonstrate machine learning is practical for identifying latent classes of distributed computation from communication patterns, without the computational overhead required for static analysis and without using a priori information.

8. Acknowledgements

Thanks to Scott Campbell and David Skinner for capturing IPM data at NERSC and to members of the high performance computing security project at LBNL for helpful discussions. 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-AC02-05CH11231, and also by the U.S. Department of Homeland Security under grant award number 2006-CS-001-000001 under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract number DE-AC02-05CH11231. The views and conclusions contained in this document are those of the authors and not necessarily those of its sponsors.

References

[1] R. Metzger and Z. Wen, *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization*. MIT Press, 2000.

[2] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Transforming MPI Source Code Based On Communication Patterns," *Future Generation Computer Systems*, vol. 26, no. 1, pp. 147–154, 2010.

[3] S. Whalen, S. Peisert, and M. Bishop, "Network-Theoretic Classification of Parallel Computation Patterns," in *Proceedings of the 1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.

[4] K. Fürlinger, N. J. Wright, and D. Skinner, "Effective Performance Measurement at Petascale Using IPM," in *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems*, pp. 373–380, 2010.

[5] 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*, 2005.

[6] R. Riesen, "Communication Patterns," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, pp. 275–282, 2006.

[7] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, "An Approach for Matching Communication Patterns in Parallel Applications," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, 2009.

[8] G. Florez-Larrahondo, Z. Liu, S. M. Bridges, A. Skjellum, and R. B. Vaughn, "Lightweight Monitoring of MPI Programs in Real Time," *Concurrency and Computation: Practice & Experience*, vol. 17, no. 13, pp. 1547–1578, 2005.

[9] A. Basumallik and R. Eigenmann, "Towards Automatic Translation of OpenMP to MPI," in *Proceedings of the 19th International Conference on Supercomputing*, pp. 189–198, 2005.

[10] A. Basumallik, S. Min, and R. Eigenmann, "Programming Distributed Memory Sytems Using OpenMP," in *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 207–214, 2007.

[11] J. Borrill, J. Carter, L. Oliker, D. Skinner, and R. Biswas, "Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms," in *Proceedings of the 2005 International Conference on Parallel Processing*, pp. 119–128, 2005.

[12] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View From Berkeley," Tech. Rep. UCB/EECS-2006-183, University of California, Berkeley, 2006.

[13] P. Colella, "Defining Software Requirements for Scientific Computing," tech. rep., DARPA High Productivity Computing Systems, 2004.

[14] T. Kohonen, "Self-Organized Formation of Topologically Correct Feature Maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.

[15] T. Kohonen, *Self-Organizing Maps*. Springer, 2000.

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

[17] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[18] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[19] K. Korb and A. Nicholson, *Bayesian Artificial Intelligence*. Chapman and Hall, 2004.

[20] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[21] C. K. Chow and C. N. Liu, "Approximating Discrete Probability Distributions with Dependence Trees," *IEEE Transactions on Information Theory*, vol. 14, no. 3, pp. 462–467, 1968.

[22] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classifiers," *Machine Learning*, vol. 29, no. 2, pp. 131–163, 1997.

[23] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley-Interscience, 1991.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[25] U. M. Fayyad and K. B. Irani, "Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 1022–1027, 1993.

[26] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd ed., 2011.

[27] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," in *Proceedings of the 23rd International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. 1381–1384, 1998.

[28] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A Supernodal Approach to Sparse Partial Pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999.

[29] S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. W. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and C. Whaley, *ScaLAPACK User's Guide*. SIAM, 1997.

[30] B. G. Pfrommer, J. W. Demmel, and H. Simon, "Unconstrained Energy Functionals for Electronic Structure Calculations," *Journal of Computational Physics*, vol. 150, no. 1, pp. 287–298, 1999.

[31] G. Kresse and J. Furthmüller, "Efficient Iterative Schemes for Ab Initio

Total-Energy Calculations Using a Plane-Wave Basis Set," *Physical Review B*, vol. 54, no. 16, pp. 11169–11186, 1996.

[32] R. Kohavi and D. H. Wolpert, "Bias Plus Variance Decomposition for Zero-One Loss Functions," in *Proceedings of the 13th International Conference on Machine Learning*, pp. 275–283, 1996.

[33] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[34] U. Alon, "Network Motifs: Theory and Experimental Approaches," *Nature Reviews Genetics*, vol. 8, no. 6, pp. 450–461, 2007.