Vrisha: Using Scaling Properties of Parallel Programs for Bug Detection and Localization

Bowen Zhou, Milind Kulkarni and Saurabh Bagchi
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN
{bzhou, milind, sbagchi}@purdue.edu

ABSTRACT

Detecting and isolating bugs that arise in parallel programs is a tedious and a challenging task. An especially subtle class of bugs are those that are *scale-dependent*: while small-scale test cases may not exhibit the bug, the bug arises in large-scale production runs, and can change the result or performance of an application. A popular approach to finding bugs is statistical bug detection, where abnormal behavior is detected through comparison with bug-free behavior. Unfortunately, for scale-dependent bugs, there may not be bug-free runs at large scales and therefore traditional statistical techniques are not viable. In this paper, we propose *Vrisha*, a statistical approach to detecting and localizing scale-dependent bugs. Vrisha detects bugs in large-scale programs by building models of behavior based on bug-free behavior at small scales. These models are constructed using kernel canonical correlation analysis (KCCA) and exploit *scale-determined* properties, whose values are predictably dependent on application scale. We use Vrisha to detect and diagnose two bugs that appear in popular MPI libraries, and show that our techniques can be implemented with low overhead and low false-positive rates.

1. INTRODUCTION

Software bugs greatly affect the reliability of high performance computing (HPC) systems. The failure data repository from Los Alamos National Lab covering 9 years till 2006, of data from 22 HPC systems, including a total of 4750 machines and 24101 processors, revealed that software was the root cause of failures between 5% and 24% of the time (depending on the system being considered) [27]. Since then, the scale of HPC systems has increased significantly and software has been asked to perform ever greater feats of agility to keep the performance numbers scaling up. As an illustrative example, consider that when the LANL data was released, the 10th ranked computer on the list of top 500 supercomputers in the world had 5,120 cores and a maximum performance (on the Linpack benchmark) of 36 TFlops. In the latest ranking, from November 2010, the 10th ranked computer has 107K cores (an increase of 20 times) and a maximum performance of 817 TFlops (an increase of 22 times). It is instructive to note that the individual processor speed in this period has increased by a factor of only 2.4 for these two sample computers.[1] Therefore, software, both at the application level and at the library level, have had to become more sophisticated to meet the scaling demands. Therefore, while more recent evidence of software failures affecting HPC systems is only for individual incidents or small-sized datasets [14, 26], we believe that it is safe to assume that software failures are playing a more important role today.

Many software bugs result in subtle failures, such as silent data corruption, some of which are detected only upon termination of the application and the rest go undetected [24], and degradation in the application performance [13]. These are undesirable because they make the results of the HPC applications untrustworthy or reduce the utilization of the HPC systems. It is therefore imperative to provide automated mechanisms for detecting errors and localizing the bugs in HPC applications. With respect to error detection, the requirement is to detect the hard-to-catch bugs while performing lightweight instrumentation and runtime computation, such that the performance of the HPC application is affected as little as possible. With respect to bug localization, the requirement is to localize the bug to as small a portion of the code as possible so that the developer can correct the bug. These two motivations have spurred a significant volume of work in the HPC community, with a spurt being observable in the last five years [6, 22, 25, 11, 10, 17, 12]. Our work also has these two goals. However, we are differentiated from existing work in the class of bugs that we focus on, which we believe closely matches bugs that are being experienced as software is being asked to scale up.

A common development and deployment scenario for HPC systems is that the developer develops the code and tests it on small(ish)-sized computing clusters. For purposes of this paper, we will consider the abstraction that the application comprises a number of processes, which communicate among themselves using some standard communication library, and the processes are distributed among the different cores in the computing system. The specific instantiation of

[1]We use the 10th-ranked computer rather than the top-ranked one because of better representativeness of the numbers for supercomputing platforms, rather than considering the outliers, which the top ones are by definition. We find a similar message when we look at the 100th-ranked supercomputer.

the abstraction could be provided by MPI, OpenMP, UPC, CAF, or others; in this paper we use MPI as the programming model. The testing done by the developer at the small scale is rigorous (for well-developed codes) in that different input datasets, architectures, and other testing conditions are tried. Both correctness and performance errors can be detected through the manual testing *as long as the error manifests itself in the small scale of the testbed that the developer is using*. For various reasons, the developer either does not do any testing at the larger scale on which the application is ultimately supposed to execute correctly, or does such testing in a far less rigorous manner. The reasons behind this include the unavailability of a large number of cores in the computing environment for testing purposes, the fact that it takes a significant amount of time for the application to run at the large scale, or that it is difficult from a human perception standpoint to keep track of executions on a large number of cores. As a result, increasingly we see errors that manifest themselves when the application is executed on a large scale. This difference in the behavior of the application between what we will call *the testing environment* and *the production environment* provides the fundamental insight that drives our current work.

The most relevant class of prior work for error detection and bug localization uses statistical approaches to create rules for correct behavior. A template for this work is that error-free runs are used to build models of correct behavior, runtime behavior is modeled using monitoring data collected at runtime by instrumenting the application or the library, and if the runtime model deviates significantly from the correct behavior model, an error is flagged. The factor that causes the difference between the two models is mapped to a code region to achieve bug localization. In the cases where error free runs are not available, the techniques make use of the assumption that the common case is correct. The common case is defined either temporally, i.e., a given process behaves correctly over most time slices, or spatially, i.e., most of the processes in the application behave correctly. We believe that for the class of bugs that we mentioned in the previous paragraph, neither of these assumptions is valid—error-free runs are not available in the production environment, and the common case may be erroneous, i.e., the bug affects all (or most of) the processes in the application and for each process, the bug manifests itself as an error in all the iterations (or equivalently, time slices).

To see an illustrative example, consider a bug in a popular MPI library from Argonne National Lab, MPICH2 [4]. The bug shown in Figure 1 is in the implementation of the MPI_Allgather routine, a routine for all-to-all communication. In MPI_Allgather, every node gathers information from every other node using different communication topologies (ring, balanced binary tree, etc.) depending on the total amount of data to be exchanged. The bug is that for a large enough scale an overflow occurs in the (internal) variable used to store the total size of data exchanged because of the large number of processes involved (see the line with the `if` statement). As a result, a non-optimal communication topology will be used. This is an instance of a performance bug, rather than a correctness bug, both of which can be handled by our proposed solution. This bug may not be evident in testing either on a small-scale system or with small amount of data.

Solution Approach.

To handle the problem of error detection and bug localization under the conditions identified above, we observe that as parallel applications scale up, *some of their properties are either scale invariant or scale determined*. By scale invariant, we mean that the property does not change as the application scales up to larger numbers of cores, and by scale determined, we mean that the property changes in a predictable manner, say in a linearly increasing manner. As an example of a scale-invariant property, we have the number of neighbors that a process communicates with and the distribution of volume of communication with its neighbors (we mention this as properties that may be present in some applications, not as a ground truth that will be present in all applications). As an example of a scale-determined property, we have the amount of data that is exchanged by a process with its communication neighbors. This observation has been made in the HPC context by previous researchers [29, 31], albeit no one has used it for error detection.

We leverage the above observation to build our system called Vrisha.[2] In it, we focus on bugs that have the above characteristic—manifest themselves at large scales—and further, the bugs affect communication behavior. For example, the bug may result in no communication with some neighbors, sending incorrect volumes of data to some, sending incorrect data types to some, sending data to a legitimate neighbor but from an incorrect context (an unexpected call site, say), etc. This is an important class of bugs because bugs of this kind are numerous, subtle, and importantly, for the distributed nature of the computation, can result in error propagation. As a result of error propagation, multiple processes may be affected, which will make it difficult to detect the failure and to perform recovery actions.

Next, we give a high-level operational flow in Vrisha. Vrisha builds a model of the application running on a small scale in the testing environment. The model is built per process and consists of communication-related parameters, such as, volume of communication with each of its neighbors, indexed by the call site. The model also includes the parameters that should determine the behavior of each process, such as, the command line arguments, the rank of the process, the total number of processes. We refer to the first part of the model as the *observational* part and the second part as the *control* part. For a deterministic application (which is our target), the premise is that there is a correlation between the control and the observational parts of the model. Such correlation may be quite complex and different for different pairs of control and observation variables. We do not make *a priori* assumptions on the nature of the correlation, e.g., we do not assume a linear relationship exists between the two parts. When the application is run at a large scale on the production testbed, the correlation should be maintained in the case of correct execution. However, in the case of an error, the correlation will be violated and Vrisha uses this as the trigger for error detection. Next, for bug localization, Vrisha determines which part of the observational model causes the correlation to break. Since the model parts are indexed by the call site, one (or a few) call site(s) can be flagged as the code regions to investigate for the purpose of fixing the bug.

For the above example of the real bug, Vrisha is able to

[2]From the Sanskrit word for the gate keeper of righteousness.

```
int MPIR_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                    int recvcount, MPI_Datatype recvtype, MPID_Comm *comm_ptr)
{
    int        comm_size, rank;
    int        mpi_errno = MPI_SUCCESS;
    int        curr_cnt, dst, type_size, left, right, jnext, comm_size_is_pof2;
    MPI_Comm   comm;
    ...
    if ((recvcount*comm_size*type_size < MPIR_ALLGATHER_LONG_MSG) && (comm_size_is_pof2 == 1)) {
    /* BUG IN ABOVE CONDITION CHECK DUE TO OVERFLOW */

        /* Short or medium size message and power-of-two no. of processes. Use
         * recursive doubling algorithm */
    }
   ...
}
```

Figure 1: Example of a real bug. This bug appears in the MPICH2 library implementation and manifests itself at a large scale.

handle it because the communication topology has a predictable evolution with scale. The bug causes the evolution pattern to be disrupted and Vrisha is able to detect it and identify the communication call where the bug lies. The diagnosis will not be down to the finest granularity of the line of code, which is not a target of our work, but to a small region of code around the communication call, whose size will depend on how frequently communication calls are present in the application. In practice, for the two case studies, this is about 20 lines of C code.

Challenges.

The above high-level approach raises several questions, which we seek to answer through the rest of the paper. What model should we use to capture the correlations in a general and yet efficient manner; we use a machine learning technique called Kernel Canonical Correlation Analysis (KCCA) [7, 28]. What communication-related features should Vrisha consider so as to handle a large class of bugs and yet not degrade the performance due to either collecting the measurement values at runtime or executing its algorithms for error detection and bug localization. Several errors manifest themselves in a process different from the one where the bug lies. How does Vrisha handle such non-local anomalies. This is challenging because if the fault-tolerance protocol requires coordination among multiple processes, that makes it likely to have a high overhead.

Thus, the contributions of this work are as follows.

1. We are the first to focus on bugs that are increasingly common as applications have to scale to larger-sized systems. These bugs manifest themselves at large scales and at these scales, no error-free run is available and the common case execution is also incorrect. This appears to be a real issue since the application will ultimately execute at these large scales and at which exhaustive testing is typically not done.
2. Our work is able to deduce correlations between the scale of execution and the communication-related properties of the application. We make no assumption about the nature of the correlaton and it can belong to one of many different classes. Violation of this correlation is indicative of an error.
3. We can handle bugs at the application level as well as at the library level because our monitoring and analysis are done at the operating system socket level, i.e., beneath the library.
4. We show through experimental evaluation that our technique is able to detect errors and localize bugs that have been reported and manually fixed prior to our work and that cannot be handled by any prior technique. We also show that in achieving this, our performance overhead is minimal (less than 8%).

The rest of the paper is organized as follows. Section 2 gives a high level description of the approach taken by Vrisha to detect and localize bugs. Section 3 describes the particular statistical technique used by Vrisha, KCCA, and explains why other statistical approaches are less suitable to our problem. Section 4 discusses the features used by Vrisha to detect errors, while Section 5 lays out the design of Vrisha in more detail, including the heuristics used to detect and localize bugs. Section 6 shows how Vrisha can be used to detect real-world bugs in MPI libraries. We wrap up with Section 7 describing prior work in fault detection, and conclude in Section 8.

2. OVERVIEW

A ten-thousand foot overview of Vrisha's approach to bug detection and diagnosis is given in Figure 2. As in many statistical bug finding techniques, Vrisha consists of two phases, the *training phase*, where we use bug-free data to construct a model of expected behavior, and the *testing phase*, where we use the model constructed in the training phase to detect deviations from expected behavior in a production run. We further subdivide the two phases into five steps, which we describe at a high level below. The phases are elucidated in further detail in the following sections.

(a) The first step in Vrisha is to collect bug-free data which will be used to construct the model. Vrisha does this by using instrumented training runs to collect statistics describing the normal execution of a program (see Section 5.1). These statistics are collected on a per-process basis. Because we are interested in the scaling behavior of a program, whether by increasing the number of processors or the input size, our training runs are conducted at multiple scales, which are still smaller than the scales of the production runs. The difficulties of doing testing or getting error-free runs at large scales that we mentioned in the Introduction, also apply to the process of building correct models and hence our training runs are done at small scales. The executions at multiple scales is meant to provide enough data to the modeling steps to allow us to capture the scaling properties of the program.

(b) After collecting profiling data from the training runs, Vrisha aggregates that data into two feature sets, the *control* set and the *observational* set. The characteristics of a par-

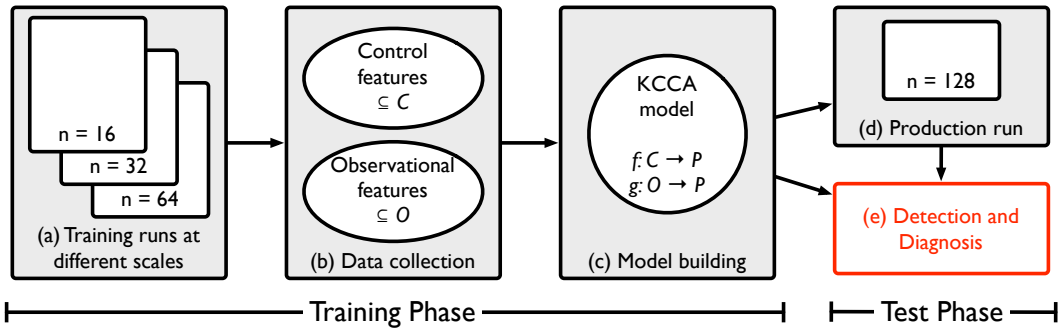


Figure 2: Overview of system architecture

ticular process in a particular training run can be described using a set of *control features*. Conceptually, these control features are the "inputs" that completely determine the observed behavior of a process. Examples of these features include the arguments to the application (or particular process) and the MPI *rank* of the process. Crucially, because we care about scaling behavior, the control features also include information on the scale of the training run, such as the number of processes in the run, or the size of the input used for the run. Each process can thus be described by a feature vector of these control features, called the *control vector*.

The control vector for a process captures the input features that determine its behavior. To describe the processes' actual behavior, Vrisha uses *observational* features that are collected at runtime through lightweight instrumentation that it injects at the socket layer under the MPI library. Example observational features for a process might include its number of neighbors, the volume of data communicated from a single call site, or the distribution of data communicated of different types. The selection of observational features constrains what types of bugs Vrisha can detect: a detectable bug must manifest in abnormal values for one or more observational features. Section 4 discusses our choice of features. The feature vector of observations for each process is called its *observation vector*.

(c) The third, and final, step of the training phase is to build a model of observed behavior. Vrisha uses KCCA [7, 28] to build this model. At a high level, KCCA learns two projection functions, $f : \mathcal{C} \to \mathcal{P}$ and $g : \mathcal{O} \to \mathcal{P}$, where $\mathcal{C}$ is the domain of control vectors, $\mathcal{O}$ is the domain of observation vectors, and $\mathcal{P}$ is a projection domain. The goal of f and g is to project control and observation vectors for a particular process into the same projection domain such that the projected vectors are within some tolerance ϵ of each other. These projection functions are learned using the control and observation vectors of bug-free runs collected in step (b).

Intuitively, if an observation vector, $o \in \mathcal{O}$, represents the correct behavior for a control vector, $c \in \mathcal{C}$, f and g should project the vectors to nearby locations in $\mathcal{P}$; if the observation vector does not adhere to expected behavior, $f(c)$ will be farther than ϵ from $g(o)$, signaling an error. Crucially, because the control vectors c include information about the program's scale, KCCA will incorporate that information into f, allowing it to capture scaling trends. Further background on KCCA is provided in Section 3. The construction of the projection functions concludes the training phase of Vrisha.

(d) To begin the testing phase, Vrisha adds instrumentation to the at-scale production run of the program, collecting both the control vectors for each process in the program, as well as the associated observation vectors. Note that in this phase we do not know if the observation vectors represent correct behavior.

(e) Finally, Vrisha performs detection and diagnosis. The control vector of each process, c, accurately captures the control features of the process, while the observation vector, o, may or may not correspond to correct behavior. Vrisha uses the projection functions f and g learned in the training phase to calculate $f(c)$ and $f(o)$ for each process. If the two projected vectors are within ϵ of each other, then Vrisha will conclude that the process's observed behavior corresponds to its control features. If the projections are distant, then the observed behavior does not match the behavior predicted by the model and Vrisha will flag the process as faulty. Vrisha then performs further inspection of the faulty observation vector and compares it to the observation vectors in the training runs, after they have been scaled up, to aid in localizing the bug. Vrisha's detection and localization strategies are described in further detail in Sections 5.3 and 5.4, respectively.

3. BACKGROUND: KERNEL CANONICAL CORRELATION ANALYSIS

In this section, we describe the statistical techniques we use to model the behavior of parallel programs, *kernel canonical correlation analysis* (KCCA) [7, 28].

3.1 Canonical Correlation Analysis

KCCA is an extension of *canonical correlation analysis* (CCA), a statistical technique proposed by Hotelling [19]. The goal of CCA is to identify relationships between two sets of variables, $\mathbf{X}$ and $\mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ describe different properties of particular objects. CCA determines two vectors $\mathbf{u}$ and $\mathbf{v}$ to maximize the correlation between $\mathbf{Xu}$ and $\mathbf{Yv}$. In other words, we find two vectors such that when $\mathbf{X}$ and $\mathbf{Y}$ are projected onto those vectors, the results are maximally correlated. This process can be generalized from single vectors to sets of basis vectors.

In our particular problem, the rows of $\mathbf{X}$ and $\mathbf{Y}$ are processes in the system. The columns $\mathbf{X}$ describe the set of "control" features of process; the set of characteristics that determine the behavior of a process in a run. For example, the features might include the number of processes in the

overall run, the rank of the particular process and the size of the input. The columns of $\mathbf{Y}$, on the other hand, capture the observed behavior of the process, such as the number of communicating partners, the volume of communication, etc. Intuitively, a row x_i of $\mathbf{X}$ and a row y_i of $\mathbf{Y}$ are two different ways of describing a single process from a training run, and CCA finds two functions f and g such that, for all i, $f(x_i)$ and $g(y_i)$ are maximally correlated. We can also think of CCA as finding two functions that map $\mathbf{X}$ and $\mathbf{Y}$ to nearby points in a common space.

3.2 Kernel Canonical Correlation Analysis

A fundamental limitation of CCA is that the projection functions that map $\mathbf{X}$ and $\mathbf{Y}$ to a common space must be linear. Unfortunately, this means that CCA cannot capture non-linear relationships between the control features and the observational features. Because we expect that the relationship between the control features and the observational features might be complex (*e.g.*, if the communication volume is proportional to the square of the input size), using linear projection functions will limit the technique's applicability.

We turn to KCCA, an extension of CCA that allows it to use *kernel functions* to transform the feature sets $\mathbf{X}$ and $\mathbf{Y}$ into higher dimensional spaces before applying CCA. Intuitively, we would like to transform $\mathbf{X}$ and $\mathbf{Y}$ using non-linear functions Φ_X and Φ_Y into $\Phi_X(\mathbf{X})$ and $\Phi_Y(\mathbf{Y})$, and apply CCA to these transformed spaces. By searching for linear relations between non-linear transformations of the original spaces, KCCA allows us to capture non-linear relationships between the two feature spaces.

Rather than explicitly constructing the higher dimensional spaces, KCCA leverages a "kernel trick" [7], allowing us to create two new matrices κ_X and κ_Y from $\mathbf{X}$ and $\mathbf{Y}$, that implicitly incorporate the higher dimensional transformation. In particular, we use a Gaussian, defining κ_X as follows:

$$\kappa_X(i,j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}$$

with κ_Y defined analogously. Because we use a Gaussian to construct the κs, when we apply CCA to κ_X and κ_Y, we effectively allow CCA to discover correlations using infinite-degree polynomials. The upshot of KCCA is that we can determine two *non-linear* functions f and g such that, for all i, the correlation between $f(x_i)$ and $g(y_i)$ is maximized. We can thus capture complex relationships between the control features and the observed features.

3.3 Comparison to Other Techniques

A natural question is why we choose to use KCCA as opposed to other model-building techniques, such as multivariate regression or principal component analysis (PCA). Multivariate regression attempts to find a function f that maps the input, independent variables X to dependent variables Y. We could consider the control features to be the independent variables, and the observational features to be the dependent variables. However, regression analysis typically requires that the input variables be independent of each other, which may not be the case. More generally, the problem with any model that simply maps the control variables to the observational variables is that such a mapping must account for all the observational variables. Consider an observational feature such as execution time, which is not truly dependent on the control features (because, *e.g.*, it is also dependent on architectural parameters that are not captured by the control features). If the model attempts to predict execution times, then it may be particularly susceptible to false positives since two non-buggy runs with the same control features may exhibit different execution times. Because KCCA projects both the control features and the observational features into new spaces, it is able to disregard features that may not be related to each other.

Another approach to modeling is to use PCA build a predictive model for the observational features. Bug detection can then be performed by seeing if the observational features of the production run correspond to the model. Unfortunately, a simple PCA-based model will not accommodate different non-buggy processes that have different observational behaviors. In particular, such a model cannot take into account scaling effects that might change the observed behavior of a program as the system size or the data size increases. Instead, additional techniques, such as clustering, would need to be applied to account for different possible behaviors. For example, we could build separate PCA models at varying scales and then apply another technique such as non-linear regression to those models to infer a function that predicts observational feature values at new scales. KCCA, by contrast, incorporates scaling effects into its modeling naturally and avoids having to separately derive a scaling model.

4. FEATURE SELECTION

A critical question to answer when using statistical methods to find bugs is, what features should we use? To answer this question, we must consider what makes for a good feature. There are, broadly, two categories of characteristics that govern the suitability of a feature for use in Vrisha: those that are necessary for KCCA to produce a scale-determined model, and those that are necessary for our techniques to be useful in finding bugs. Furthermore, because Vrisha uses KCCA to build its models, we must concern ourselves with both control features and observational features.

First, we consider what qualities a feature must possess for it to be suitable for Vrisha's KCCA-based modeling.

- The control features we select must be related to the observational features we collect. If there is no relation, KCCA will not be able to find a meaningful correlation between the control space and the observation space. Moreover, because we care about scaling behavior, the scale (system and input size) must be included among the control features.
- The observational features should be scale-determined: Changing the scale while holding other control features constant should either have no effect on behavior or affect behavior in a deterministic way. Otherwise, Vrisha's model will have no predictive power.

Second, we consider what criteria a feature must satisfy for it to provide useful detectability.

- The control features must be easily measurable. Our detection technique (described in Section 5.3) assumes that the control vector for a potentially-buggy test run is correct.
- The observational features must be efficient to collect. Complex observational features will require instrumen-

tation that adds too much overhead to production runs for Vrisha to be useful.

- The observational features must be possible to collect without making any change to the application. This is needed to support existing applications and indicates that the instrumentation must be placed either between the application and the library, or under the library. Vrisha's instrumentation is placed under the library.
- The observational features must reflect any bugs that are of interest. Vrisha detects bugs by finding deviations in observed behavior from the norm. If the observational features do not change in the presence of bugs, Vrisha will be unable to detect faults. Notably, this means that the types of bugs Vrisha can detect are constrained by the choice of features.

Features used by Vrisha .

The features used by Vrisha consist of two parts corresponding to the control feature set $\mathcal{C}$ and the observational feature set $\mathcal{O}$. The control features include (a) the process ID, specifically Vrisha uses the rank of process in the default communicator `MPI_COMM_WORLD` because it is unique for each process in the same MPI task; (b) the number of processes running the program, which serves as the scale parameter to capture system scale-determined properties in communication; (c) the argument list used to invoke the application, which serves as the parameter that correlates with input scale-determined properties in the communication behavior of application because it typically contains the size of the input data set.

On the other hand, the observational feature set of the i^{th} process is a vector $\mathbf{D_i}$ of length c, where c is the number of distinct MPI call sites manifested in one execution of the program.

$$\mathbf{D_i} = (d_{i1}, \cdots, d_{ic})$$

The j^{th} component in $\mathbf{D_i}$ is the volume of data sent at the j^{th} call site. The index j of call sites has no relation with the actual order of call sites in the program. In fact, we uniquely identify each call site by the call stack it corresponds. The observational features capture the aggregated communication behavior of each process.

The set of control and observational features we choose has several advantages. First, they are particularly suitable for our purpose of detecting communication-related bugs in parallel programs. Second, it is possible to capture these features with a reasonable overhead so they can be instrumented in production runs. These features are easy to collect through instrumentation at the Socket API level. Further, with these features, we can identify call sites in a buggy process that deviate from normal call sites and further to localize the potential point of error by comparing the call stacks of the buggy process and the normal process.

We could also add to this, the features from previous solutions. For example, the frequent-chain and chain-distribution features from DMTracker [17] are good candidates to be adapted into the observational variable set in Vrisha's framework. Also, the distribution of time spent in a function used by Mirgorodskiy *et al.* [25] is also a good feature to characterize timing properties of functions in a program and can also be imported into Vrisha to diagnose performance-related bugs as in prior work.

5. DESIGN

In this section, we explain the design of the runtime profiling component, the KCCA prediction model, bug detection method and bug localization method in Vrisha.

5.1 Communication Profiling

In order to detect bugs in both application and library level, we implement our profiling functionality below the network module of the MPICH2 library and on top of the OS network interface. So the call stack we recorded at the socket level would include functions from both the application and the MPICH2 library. The call stack and volume of data involved in each invocation of the underlying network interface made by MPICH2 is captured and recorded by our profiling module. This design is distinct from FlowChecker where, though the instrumentation is at the same layer as ours, it can only capture bugs in the library. Thus, application-level calls are not profiled at runtime by FlowChecker.

5.2 Using KCCA to Find the Scaling Direction of Communication

5.2.1 Build KCCA Model

First, we need to construct the two square kernel matrices from the values of the control and the observational variables respectively. These matrices capture the similarity in the values of one vector with another. Thus, the cell (i, j) will give the numerical similarity score between vector (control or observational) i and vector j. Since all our variables are numerical, we use the Gaussian kernel function [28] to create the kernel matrices, which is defined as follows:

$$\kappa_{Gaussian}(y_i, y_j) = e^{-\frac{\|\mathbf{y}_i - \mathbf{y}_j\|^2}{\sigma_y^2}}$$

where $\|\mathbf{y}_i - \mathbf{y}_j\|$ defines the Euclidean distance and σ_y is calculated based on the variance of the norms of the data points. If two vectors are identical then the kernel function will give a score of 1. Then we solve the KCCA problem to find the projections from the two kernel matrices into the projection space that give the maximal correlation of the control and the observational variables in the training sets. Finally, we can use the solution of KCCA to project both control and observational variables to the same space spanned by the projection vectors from KCCA.

5.2.2 Parameters in the KCCA Model

As done in previous work [15, 16], we set the inverse kernel width σ in the Gaussian kernel used by KCCA to be a fixed fraction of the sample variance of the norms of data points in the training set. Similarly, we used a constant value as the regularisation parameter γ throughout all our experiments. We also give a preliminary study on the sensitivity of Vrisha to the model selection of KCCA in Section 6.4 for the sake of completeness.

5.3 Using Correlation to Detect Errors

To detect if there is an error, we use for each process the correlation between control vector and the observational vector in the projected space spanned by the projection vectors that are available from the solution of KCCA. The lack of

correlation is used as a trigger for detection and the quantitative value of correlation serves as the metric of abnormality of each process. Since KCCA provides two projection vectors that maximizes correlation between the control and observational variables, most normal processes would have a relatively high correlation between the two sets of variables. Therefore, we can set a threshold on the deviation of correlation from 1 (which corresponds to perfectly correlated) to decide whether a process is normal or abnormal. Thus, implicitly, our detection strategy achieves localization of the problem manifestation to the process level.

5.4 Localization of Bugs

5.4.1 Bugs that do not Cause Application Crash

Our strategy for localization of bugs uses the premise that the communication behavior in the production run should look similar to that in the training runs, after normalizing for the scale. The similarity should be observed at the granularity of the call sites, where the relevant calls are those that use the network socket API under the MPI library. So the localization process proceeds as follows. Vrisha matches up the call sites from the training runs and the production run in terms of their communication behavior and orders them by volume of communication. For example, in Figure 3, the matches are (call site ID in training, call site ID in production): (2, 3), (1, 2), (3, 1), (4, 4). The call site ID order does not have any significance, it is merely a map from the call stack to a numeric value. Now for the matching call sites, the call stacks should in the correct case be the same, indicating that the same control path was followed. A divergence indicates the source of the bug. Vrisha flags the points in the call stack in the production run where it diverges from the call stack in the training run, starting from the bottom of the call stack (i.e., the most recent call). The call stack notation is then translated back to the function and the line number in the source code to point the developer to where she needs to look for fixing the bug.

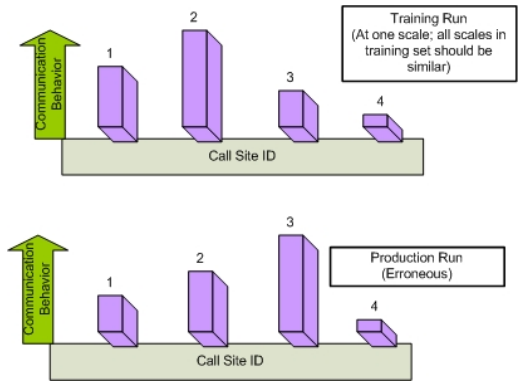


Figure 3: Example for demonstrating localization of a bug.

As would be evident to the reader, Vrisha determines an ordered set of code regions for the developer to examine. In some cases, the set may have just one element, namely, where there is only one divergent call site and only one divergence point within the call site. In any case, this is helpful to the developer because it narrows down the scope of where she needs to examine the code.

Retrieving Debugging Information. To facilitate the localization of bugs, we need certain debugging information in executables and shared libraries to map an address A in the call stack to function name and offset. In case such information is stripped off by the compiler, we also need to record the base address B of the object (executable and shared library) when it is loaded into memory so the offset within the object $A - B$ can be calculated and translated into the function name and the line number. This is done in an off-line manner, prior to providing the information to the developer for debugging, and can be done by an existing utility called `addr2line`.

5.4.2 Bugs that Cause Application Crash

It is trivial to detect an error caused by a bug that makes the application crash. However, localization of the root cause of such bugs is not as easy. For this, we use the localization technique for non-crashing bugs as the starting point and modify it. For comparison with the communication behavior of the training runs, we identify the point in execution corresponding to the crash in the production run. We then eliminate all call sites in the training run after that point from further processing. Then we follow the same processing steps as for the non-crashing bugs. One distinction is in the way we order the different call sites. The call site which is closest to the point at which the application crashed is given the highest priority. The intuition is that the propagation distance between the bug and the error manifestation is more likely to be small than large. Hence, we consider the call stack from the crashed application (in the production run) and compare that first to the call stack from the closest point in the training runs and flag the points of divergence, starting from the latest point of divergence.

5.5 Discussion

Our proposed design for Vrisha has some limitations, some of which are unsurprising, and some of which are somewhat subtle. The most obvious limitation is that Vrisha's ability to detect bugs is constrained by the choice of features. This limitation is imposed by the observational features and, surprisingly, the control features. If a bug manifests in a manner that does not change the value of an observational feature, Vrisha will be unable to detect it, as there will be no data that captures the abnormal behavior. Hence, the observational features must be chosen with some care to ensure that bugs are caught. Interestingly, the control features must be chosen carefully, as well. Our technique detects bugs when the expected behavior of a process (as determined by its control features) deviates from its observed behavior (as determined by its observational features). If an observational feature (in particular, the observational feature where a bug manifests) is uncorrelated with any of the control features, KCCA will ignore its contribution when constructing the projection functions and hence Vrisha will be unable to detect the bug.

Another limitation that is unique to our choice of KCCA as Vrisha's modeling technique is that KCCA is sensitive to the choice of kernel functions. As an obvious example, if the kernel function were linear, KCCA would only be able to apply linear transformations to the feature sets before finding correlations, and hence would only be able to extract linear relationships. We mitigate this concern by using a Gaussian as our kernel function, which is effectively an infinite-degree

polynomial.

Our localization strategy is also limited by the localization heuristics we use. First, we must infer a correspondence between the features of the buggy run and the features of the non-buggy runs. In the particular case of call-stack features, this presents problems as the call stacks are different for buggy vs. non-buggy runs. Our matching heuristic relies on the intuition that while the volume of data communicated at each call site is scale-determined, the *distribution* of that data is *scale invariant* (*i.e.*, is the same regardless of scale). This allows us to match up different call sites that nevertheless account for a similar proportion of the total volume of communication. While this heuristic works well in practice, it will fail if the distribution of communication is not scale-invariant. Another drawback of our localization heuristic is that if several call sites account for similar proportions of communication, we will be unable to localize the error to a single site; instead, we will provide some small number of sites as candidates for the error.

6. EVALUATION

In this section, we evaluate the performance of Vrisha against real bugs in parallel applications. We use the MPICH2 library [4] and NAS Parallel Benchmark Suite [8] in these experiments. We have augmented the MPICH2 library with communication profiling functionality and reproduce reported bugs of MPICH2 to test our technique. The NAS Parallel Benchmark Suite 3.3 MPI version is used to evaluate the runtime overhead of the profiling component of Vrisha.

The experiments show that Vrisha is capable of detecting and localizing realistic bugs from the MPICH2 library while its runtime profiling component incurs less than 10% overhead in tests with the NAS Parallel Benchmark Suite. We also compare Vrisha with some of the most recent techniques for detecting bugs in parallel programs and illustrate that the unique ability of Vrisha to model the communication behavior of parallel programs as they scale up is the key to detect the evaluated bugs.

All the experiments are conducted on a 15 node cluster running Linux 2.6.18. Each node is equipped with two 2.2GHz AMD Opteron Quad-Core CPUs, 512KB L2 cache and 8GB memory.

6.1 Allgather Integer Overflow in MPICH2

6.1.1 Description

This bug is an integer overflow bug which causes MPICH2 to choose a performance-suboptimal algorithm for Allgather (Figure 1). Allgather is a collective all-to-all communication function defined by the MPI standard, in which each participant node contributes a piece of data and collects contributions from all the other nodes in the system. Three algorithms [30] are employed to implement this function in the MPICH2 library and the choice of algorithm is conditioned on the total amount of data involved in the operation.

The total amount of data is computed as the product of three integer variables and saved in a temporary integer variable. When the product of the three integers overflows the size of an integer variable, a wrong choice of the algorithm to perform Allgather is made and this results in a performance degradation, which becomes more significant as the system scales up. The bug is more likely to happen on a large-scale system, i.e., with a large number of processors, because one

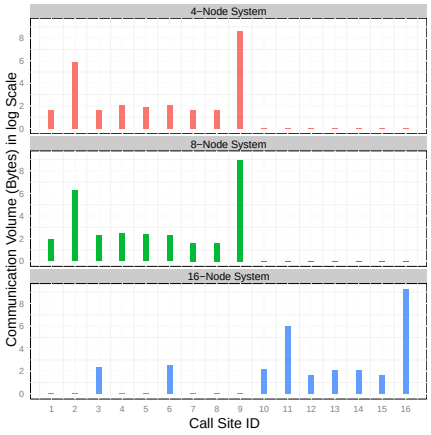


Figure 4: Communication behavior for the Allgather bug at two training scales (4 and 8 nodes) and production scale system (16 nodes). The bug manifests itself in the 16 node system (and larger scales)

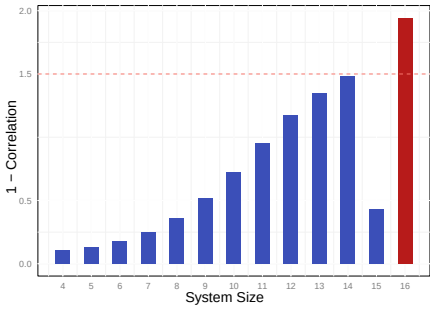


Figure 5: Correlation in the projection space using the KCCA-generated maps for systems of different scale. The low correlation value at scale 16 indicates the error.

of the multiplier integer is the number of processes calling Allgather. For example, on a typical x86_64 Linux cluster with each process sending 512 KB of data, it will take at least 1024 processes to overflow an integer.

The bug has been fixed in a recent version of MPICH2 [1]. However, we found a similar integer overflow bug in Allgatherv, a variant of Allgather to allow varying size of data contributed by each participant, still extant in the current version of MPICH2 [2].

6.1.2 Detection and Localization

For the ease of reproducing the bug, we use a simple synthetic application that does collective communication using Allgatherv and run this application at increasing scales. The test triggers the bug in the faulty version of MPICH2 if the number of processes is 16 or more. Vrisha is trained with the communication profiles of the program running on 4 to 15 processes where the bug is latent and the communication distribution is not contaminated by the bug. We pictorially represent in Figure 4 the communication behavior that is seen in the application for two different sizes of the training system (4 and 8 processes) and one size of the production system where the bug manifests itself (16 processes). The X-axis is the different call sites (the IDs do not have any significance, they are numerical maps of the call stacks) and the Y-axis is the volume of communication, which is used

```
Call Stack 9:                          Call Stack 16:
============================           ============================
MPID_nem_tcp_send_queued+0x1cc         MPID_nem_tcp_send_queued+0x1cc
MPID_nem_tcp_connpoll+0x3a3            MPID_nem_tcp_connpoll+0x3a3
MPID_nem_network_poll+0x1e             MPID_nem_network_poll+0x1e
MPIDI_CH3I_Progress+0x2ab              MPIDI_CH3I_Progress+0x2ab
MPIC_Wait+0x89                         MPIC_Wait+0x89
MPIC_Sendrecv+0x246                    MPIC_Sendrecv+0x246
MPIR_Allgatherv+0x6a2  <------>        MPIR_Allgatherv+0x17fd
PMPI_Allgatherv+0x1243                 PMPI_Allgatherv+0x1243
main+0x14c                             main+0x14c
__libc_start_main+0xf4                 __libc_start_main+0xf4
```

Figure 6: Call stacks for the correct case (call stack 9, in the training system) and the erroneous case (call stack 16, in the production system). The deepest point of the stack corresponding to the last called function is at the top of the figure.

in this illustration as a representative of the communication behavior normalized to the scale of the system. The divergence in the communication behavior shows up with 16 processes where the pattern of communication behavior looks distinctly different. Vrisha successfully detects this bug as the correlation in the projection space for the 16-node system is low, as depicted in Figure 5. The Y-axis is the inverse of the correlation, so a high value there indicates low correlation. The cutoff is set such that there is no false positive in the training set and that is sufficient for detecting the error. Note that this bug affects *all* processes at systems of size 16 or higher and therefore, many previous statistical machine learning techniques will not be able to detect this because they rely on majority behavior being correct.

According to our bug localization scheme, Vrisha compares the normal and the faulty distributions in Figure 4. The call site 9 from the training run is matched up with call side 16 from the production run and this is given the highest weight since the communication volume is the largest (90% of total communication). We show the two call stacks corresponding to these two call sites in Figure 6. The deepest point in the call stack, i.e., the last called function, is shown at the top in our representation. A comparison of the two call stacks reveals that the faulty processes take a detour in the function `MPIR_Allgatherv` by switching to a different path. The offset is mapped to line numbers in the `MPIR_Allgatherv` function and a quick examination shows that a different conditional path is taken for an `if` statement. The condition for the `if` statement is the temporary integer variable that stores the total amount of data to be transmitted by Allgather. This is where the overflow bug lies.

6.1.3 *Comparison with Previous Techniques*

This bug cannot be detected by previous techniques [25, 17, 11, 20] which capture anomalies by comparing the behavior of different processes in the same sized system. This is due to the fact that there is no statistically significant difference among the behaviors of processes in the 16-node system. As the bug degrades the performance of Allgather but no deadlock is produced, those techniques targeted at temporal progress [6] will not work either. Finally, since there is no break in the message flow of Allgather as all messages are delivered eventually but with a suboptimal algorithm, FlowChecker [12] will not be able to detect this bug. Therefore, Vrisha is a good complement to these existing techniques for detecting subtle scale-dependent bugs in parallel programs.

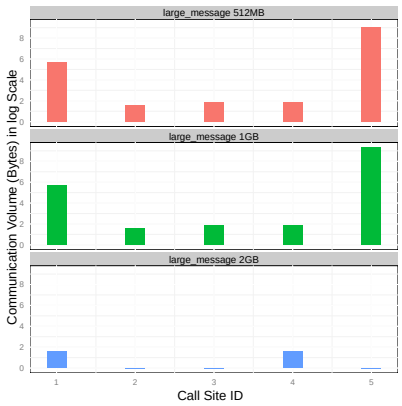


Figure 7: Communication behavior for the large message bug at two training scales (512 MB and 1 GB) and production scale system (2 GB). The bug manifests itself in data sizes of 2 GB and larger.

6.2 Bug in Handling Large Messages in MPICH2

6.2.1 *Description*

This bug [3] was first found by users of PETSc [9], a popular scientific toolkit built upon MPI. It can be triggered when the size of a single message sent between two physical nodes (not two cores in the same machine) exceeds 2 gigabytes. The MPICH2 library crashes after complaining about dropped network connections.

It turns out that there is a hard limit on the size of message can be sent in a single iovec struct from the Linux TCP stack. Any message that violates this limit would cause socket I/O to fail as if the connection were dropped. The most tricky part is that it would manifest in the MPI level as a MPICH2 bug to the application programmers.

6.2.2 *Detection and Localization*

Since we have no access to the original PETSc applications that triggered this bug, we compromise by using the regression test of the bug as our data source to evaluate Vrisha against this bug. The regression test, called `large_message`, is a simple MPI program which consists of one sender and two receivers and the sender sends a message a little bit larger than 2GB to each of the two receivers. We adapt `large_message` to accept an argument which specifies the size of message to send instead of the hard-coded size in the original test so we can train Vrisha with different scales of input. Here, "scale" refers to the size of data, rather than the meaning that we have been using so far—number of processes in the system. This example points out the ability of Vrisha to deduce behavior that depends on the size of data and to perform error detection and bug localization based on that. We first run the regression test program with 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, and 1GB to get the training data set and then test with the 2GB case. The distributions of communication over call sites of a representative process in each case of 512MB, 1GB, and 2GB are shown in Figure 7.

Since the bug manifests as a crash in the MPICH2 library, there is nothing left to be done with the detection part. We are going to focus on explaining how we localize the bug with the guidance from Vrisha. First of all, as discussed in Section 5.4.2, we need the stack trace at the time of the crash.

```
Call Stack 5                          Crash Stack from MPICH2
============================          ============================
MPID_nem_tcp_send_queued+0x132        MPID_nem_tcp_send_queued
(Unknown static function)             state_commrdy_handler
MPID_nem_tcp_connpoll+0x3d8           MPID_nem_tcp_connpoll
MPID_nem_network_poll+0x1e            MPID_nem_network_poll
(Unknown static function)             MPID_nem_mpich2_blocking_recv
MPIDI_CH3I_Progress+0x1d8             MPIDI_CH3I_Progress
MPI_Send+0x8ff                        MPI_Send
main+0x121)
__libc_start_main+0xf4)
```

Figure 8: Call stacks from a normal process (left) and at the point of crash due to large-sized data. Error message "socket closed" reported by MPICH2 at `MPID_nem_tcp_send_queued` helps localize the bug.

This is shown on the right part of Figure 8. In fact, the MPICH2 library exits with error message "socket closed" at function `MPID_nem_tcp_send_queued`. Comparing with all the five normal call stacks shown in Figure 7 (i.e., obtained from training runs), we find call stack 5 is almost a perfect match for the crash stack trace from MPICH2 except for two static functions whose names are optimized out by the compiler. The first divergent point in the crash trace is at `MPID_nem_tcp_send_queued`, which is where the bug lies.

To this point, we have localized the bug to a single function. The next step depends on the properties of each specific bug. In practice, most applications implement some error handler mechanism that provide useful error messages before exiting. In the case of this bug, one only needs to search for the error message "socket closed" inside the function `MPID_nem_tcp_send_queued` and would find that it is the failure of `writev` (a socket API for sending data over the underlying network) that misleads MPICH2 to think the connection is closed. In this case, Vrisha only has to search within a single function corresponding to the single point of divergence. In more challenging cases, Vrisha may have to search for the error message in multiple functions. In the absence of a distinct error message, Vrisha may only be able to provide a set of functions which the developer then will need to examine to completely pinpoint the bug.

6.2.3 *Comparison with Previous Techniques*

Most previous techniques based on statistical rules will not be helpful in localizing this bug because they lack the ability to derive scale-parametrized rules to provide role model to compare with the crash trace. All the processes at the large data sizes suffer from the failure and therefore contrary to the starting premise of much prior work, majority behavior itself is erroneous. However, FlowChecker is capable of localizing this bug since the message passing intention is not fulfilled in `MPID_nem_tcp_send_queued`.

6.3 Performance Measurement

This section studies the runtime overhead caused by Vrisha's runtime profiling in five representative applications from the NAS Parallel Benchmark Suite running on top of the MPICH2 library. The five benchmarks used in this study are CG, EP, IS, LU, and MG. Each application is executed 10 times and the average running time is used to calculate the percentage overhead due to the profiling conducted at runtime by Vrisha. All experiments are done with 16 processes and class A of NAS Parallel Benchmarks. As shown in Figure 9, the average overhead incurred by the profiling of Vrisha in the five benchmarks are all less than 8%. Note

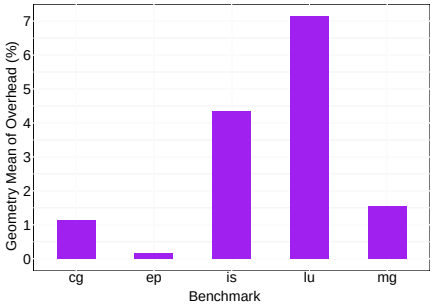


Figure 9: Overhead due to profiling in Vrisha for NASPAR Benchmark applications.

Table 1: Sensitivity of False Positive Rate to Model Parameters in Vrisha

Parameter	Range	False Positive
N_{comps}	$1, \cdots, 10$	2.85%, 3.16%
γ	$2^{-20}, \cdots, 2^{0}$	2.32%, 3.25%
σ_x	$2^{-20}, \cdots, 2^{20}$	1.79%, 8.19%
σ_y	$2^{-20}, \cdots, 2^{20}$	2.18%, 4.01%

the overhead shown here does not include the off-line procedures of model building, error detection and localization since the runtime overhead is the most important criterion to decide whether a debugging technique is practical or not while the cost of analysis can be tolerated by developers as long as it can provide useful information to speed up the process of debugging.

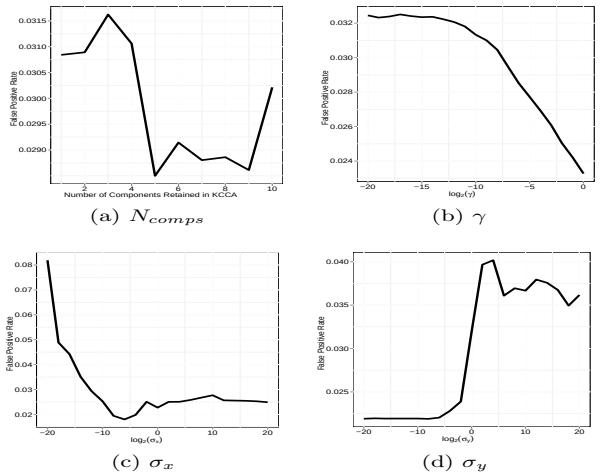
6.4 Model Selection and False Positive Rate

This section evaluates the impact of model selection of the KCCA method on the false positive rate of Vrisha. Because of the lack of a publicly-available comprehensive database of bugs in parallel programs, we have no way to conduct a study of false negative rate, therefore we follow the practice of previous researchers of focusing on the fault positive rate of our model by considering error-free applications.

The following parameters in the KCCA model, N_{comps}, γ, σ_x, σ_y are measured using five-fold cross validation on the training data from Section 6.1. The range of parameters used in the study is shown in Table 1. Note here the fractional factors instead of the actual values of σ_x and σ_y are listed in the table. We test with all different combinations of values of these parameters and calculate a curve of average false positive rate for each individual parameter as displayed in Figure 10. According to the results, N_{comps}, γ and σ_y do not significantly affect the false positive rate while σ_x has more impact taking the false positive to 8.2% in the worst case. Since this represents the worst case in terms of the worst selection of all model parameters, 8.2% is a pessimistic upper bound for the false positive rate of Vrisha. Overall, the KCCA model is not very sensitive to parameter selection which makes it more accessible to users without solid background in machine learning.

7. RELATED WORK

One way of classifying existing work on error detection and bug localization is whether invariants are expected to hold in a deterministic manner or stochastically. In the first class, some property to be validated at runtime is inserted as an invariant and the invariant must hold true in all ex-



(a) N_{comps} (b) γ (c) σ_x (d) σ_y

Figure 10: Average False Positive Rate of KCCA Models w.r.t. Each Parameter by 5-fold Cross Validation

ecutions [12, 18]. In the second class, the property must hold statistically [22, 11, 22, 10, 11]. A typical example is that if the behavior of a process *over an aggregate* is similar to the aggregate behavior of a large number of other processes, then the behavior is considered correct. Our work adopts a statistics-based approach. Below we review related work that is aimed at error detection and bug localization for parallel applications, for different classes of errors.

The first work in this domain that illuminates our work is that by Mirgorodskiy *et al.* [25], which applies to similarly behaving processes in an application. The system creates a feature vector for each process, where each element of the vector is the fraction of time spent in a particular function. Under normal execution, all processes' feature vectors should be similar. Errors are detected using one of two methods. In the first method, supervised data is available, indicating correct behavior; a process is flagged as anomalous if its vector is far from any normal trace. In the absence of supervised data, processes are flagged as anomalous if its distance from the kth nearest neighbor is too large. Localization is performed by determining which entry in the anomalous vector contributed most to its distance, indicating a particular function responsible for the misbehavior.

The second relevant work in this domain is AutomaDeD [11]. This work provides a model to characterize the behavior of parallel applications. It models the the control flow and timing behavior of application tasks as Semi-Markov Models (SMMs) and detects faults that affect these behaviors. SMM states represent Given an erroneous execution of the application, AutomaDeD examines how each task's SMM changes over time and relates to the SMMs of other tasks. First, AutomaDeD detects which time period in the execution of the application is likely erroneous. AutomaDeD then clusters task SMMs of that period and performs cluster isolation, which uses a novel similarity measure to identify the task(s) suffering from the fault. Finally, transition isolation detects the transitions that were affected by the fault more strongly or earlier than others, thus possibly identifying the buggy code region.

The third and fourth pieces of work — DMTracker [17] and FlowChecker [12] — fall in the same class, namely, for handling bugs related to communication. DMTracker uses data movement related invariants, tracking the frequency of data movement and the chain of processes through which data moves. The premise of DMTracker is that these invariants are consistent across normal processes. Bugs are detected when a process displays behavior that does not conform to these invariants, and can be localized by identifying where in a chain of data movements the invariant was likely to be violated.

FlowChecker focuses on communication-related bugs in MPI libraries. It argues that statistics-based approaches have limitations in that they cannot detect errors which affect all processes or only a small number of processes. Therefore, it uses deterministic invariants, analyzing the application-level calls for data movement to capture patterns of data movement (*e.g.*, by matching MPI_Sends with MPI_Receives). At run-time, it tracks data movement to ensure that it conforms to the statically-determined models. Localization follows directly: the data movement function that caused the discrepancy from the static model is the location of the bug.

Our work builds on the lessons from the solutions surveyed above. However, distinct from existing work, it squarely targets the way parallel applications are being designed, developed, and deployed on large scales. Thus, we aim to deduce properties from executions of the application on a small scale (as the developer may do on her development cluster) and use those properties for bug detection and localization at a large scale. Further, our work is geared to handling bugs that affect many (or all) processes in the application, as may happen in Single Program Multiple Data (SPMD)-type applications. None of the approaches above except FlowChecker are suitable for this class, while FlowChecker targets a narrow bug class - only communication bugs in the libraries and only correctness bugs rather than performance bugs. Vrisha does not have these restrictions.

More tangentially related to our work is volumes of work done in the area of general software bug detection, which includes use of program assertions, static analysis, dynamic checking, model checking, and formal verification. While the lessons learned in these influence our work, they are not directly relevant to large-scale parallel applications.

Problem diagnosis in large systems mainly focuses on isolating the root causes of system failures or performance problems and influences our design of the bug localization algorithm. Most existing studies [5, 21, 23, 25] utilize machine learning or statistical methods to study error propagation or identify program anomalies. These methods provide useful hints on diagnosing system problems. For example, Maruyama and Matsuoka propose comparing function traces from normal runs to those of failed runs for fault localization [23]. Unlike these approaches, Vrisha exploits the semantics of communication behavior as the system scales up to localize the bugs.

8. CONCLUSION

In this paper, we introduced Vrisha, a framework for detecting bugs in large-scale systems using statistical techniques. While prior work based on statistical techniques relied on the availability of error-free training runs at the same scale as production runs, it is infeasible to use full-scale systems for development purposes. Unfortunately, this means that prior bug-detection techniques are ill-suited to dealing with bugs that only manifest at large scales. Vrisha was de-

signed to tackle precisely these challenging bugs. By exploiting *scale-determined* properties, Vrisha uses kernel canonical correlation analysis to build models of behavior at large scale by generalizing from small-scale behavioral patterns. Vrisha incorporates heuristics that can use these extrapolated models to detect and localize bugs in MPI programs. We studied two bugs in the popular MPICH2 communication library that only manifest as systems or inputs scale. We showed that Vrisha could automatically build sufficiently accurate models of large-scale behavior such that its heuristics could detect and localize these bugs, without ever having access to bug-free runs at the testing scale. Furthermore, Vrisha is able to find bugs with low instrumentation overhead and low false positive rates. In further work, we will consider other kinds of bugs beyond communication-related bugs, investigate more fully the scaling behavior with respect to data sizes, and evaluate the scalability of the detection and the localization procedures.

9. REFERENCES

[1] `https://trac.mcs.anl.gov/projects/mpich2/changeset/5262`.

[2] `https://trac.mcs.anl.gov/projects/mpich2/browser/mpich2/trunk/src/mpi/coll/allgatherv.c`.

[3] `http://trac.mcs.anl.gov/projects/mpich2/ticket/1005`.

[4] The MPICH2 Project. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 74–89, 2003.

[6] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, 2009.

[7] F. R. Bach and M. I. Jordan. Kernel independent component analysis. *J. Mach. Learn. Res.*, 3:1–48, March 2003.

[8] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. Nas parallel benchmark results. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[9] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2009. `http://www.mcs.anl.gov/petsc`.

[10] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, , M. Schulz, and D. H. Ahn. Statistical Fault Detection for Parallel Applications with AutomaDeD. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, pages 1–6, 2010.

[11] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, , and M. Schulz. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 231–240, June-July 2010.

[12] Z. Chen, Q. Gao, W. Zhang, and F. Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *Proceedings of the 2010 ACM/IEEE International Conference on Supercomputing*, SC '10, pages 1–11, 2010.

[13] N. DeBardeleben. Fault-Tolerance for HPC at Extreme Scale, 2010.

[14] S. Fu and C. Xu. Exploring Event Correlation For Failure Prediction In Coalitions Of Clusters. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12. ACM, 2007.

[15] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 1–6, 2009.

[16] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 592–603, 2009.

[17] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 1–12, 2007.

[18] D. Herbert, V. Sundaram, Y.-H. Lu, S. Bagchi, and Z. Li. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. *ACM Trans. Auton. Adapt. Syst.*, 2, September 2007.

[19] H. Hotelling. Relations between two sets of variates. *Biometrika*, 28(3/4):pp. 321–377, 1936.

[20] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *FAST'10: Proceedings of the 8th USENIX conference on File and storage technologies*, pages 1–14, 2010.

[21] Z. Lan, Z. Zheng, and Y. Li. Toward automated anomaly identification in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 21:174–187, 2010.

[22] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC)*, SC '08, pages 1–9, 2008.

[23] N. Maruyama and S. Matsuoka. Model-based fault localization in large-scale computing systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, 2008.

[24] S. Michalak. Silent Data Corruption: A Threat to Data Integrity in High-End Computing Systems. In *Proceedings of 2009 National HPC Workshop On Resilience*, 2009.

[25] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[26] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Prediction of resource availability in fine-grained cycle sharing systems and empirical evaluation. *Journal of Grid Comput.*, 5(2):173–195, 2007.

[27] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *International Conference on Dependable Systems and Networks (DSN)*, pages 249 –258, 2006.

[28] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

[29] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Performance Modeling and Prediction. In *ACM/IEEE Conference on Supercomputing*, pages 1–17, 2002.

[30] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49, 2005.

[31] X. Wu and F. Mueller. ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Program. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10, 2011.