AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks

Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong Ahn, Martin Schulz

March 24, 2010

International Conference on Dependable Systems and Networks
Chicago, IL, United States
June 28, 2010 through July 1, 2010

AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks [*]

Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, and Martin Schulz

Abstract

Today's largest systems have over 100,000 cores, with million-core systems expected over the next few years. This growing scale makes debugging the applications that run on them a daunting challenge. Few debugging tools perform well at this scale and most provide an overload of information about the entire job. Developers need tools that quickly direct them to the root cause of the problem. This paper presents AutomaDeD, a tool that identifies which tasks of a large-scale application first manifest a bug at a specific code region at a specific point during program execution. AutomaDeD creates a statistical model of the application's control-flow and timing behavior that organizes tasks into groups and identifies deviations from normal execution, thus significantly reducing debugging effort. In addition to a case study in which AutomaDeD locates a bug that occurred during development of MVAPICH, we evaluate AutomaDeD on a range of bugs injected into the NAS parallel benchmarks. Our results demonstrate that detects the time period when a bug first manifested itself with 90% accuracy for stalls and hangs and 70% accuracy for interference faults. It identifies the subset of processes first affected by the fault with 80% accuracy and 70% accuracy, respectively and the code region where where the fault first manifested with 90% and 50% accuracy, respectively.

1 Introduction

The number of cores used in large scale systems already exceeds million cores in the near future [7]. As a result, the challenge of developing correct, high performance applications is also growing. When an application does not complete or completes with incorrect results, the developer must identify the offending MPI task and then the portion of the code in that task that caused the error. Traditional parallel debugging tools [12, 14, 18, 23] often perform poorly at large task counts. We focus on developing a detection tool set that identifies the offending task and, to a customizable granularity, the relevant portion of code within the task.

We present AutomaDeD, a tool set that achieves this goal of focusing debugging efforts to improve developer efficiency. It performs runtime monitoring of a parallel application to build a statistical model of the application's typical timing and control flow behavior. The typical use case for AutomaDeD is that a user suspects a run of an application is erroneous and would like to get some guidance to what parts of the application code to focus on for debugging. AutomaDeD achieves this by identifying the period in time, the task(s), and the *error site*, the region of code, where a fault first manifests itself. Thus, AutomaDeD provides the basis for eventual root cause diagnosis including identification of the exact erroneous line of source code.

This paper makes technical contributions in *two broad areas*. First, we describe *a model to characterize the behavior of parallel applications*. Second, we present methods that *compare the behavior of tasks in a parallel application in time and in space* to identify the error site. AutomaDeD models the the control flow and timing behavior of application tasks as *Semi-Markov Models (SMMs)* and detects faults that affect these behaviors. States of these SMMs represent regions of application code and edges represent execution progress from one region to another. SMMs capture the probability of transitioning from one region to another and the distribution of times spent in each region. We delimit code regions by MPI calls and use MPI calls (along with call stack information) and the computation interleaved between them as two different kinds of states in SMMs.

AutomaDeD examines how each task's SMM changes over time and relates to the SMMs of other tasks to identify the task and code region where a given fault is first manifested. 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 identifying the code region where the fault is first manifested. In addition to focusing the developer on the root cause of their bug, AutomaDeD also enables the use of traditional

[*]This work partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-xxxxxx). The work of the Purdue authors was partially supported by the National Science Foundation under grant CSR-0916337.

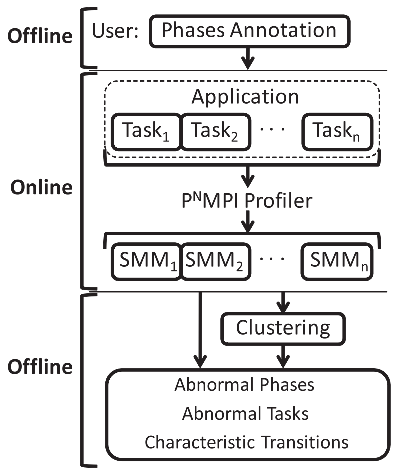

Figure 1. Design of AutomaDeD

debuggers, such as gdb and TotalView [23] at previously infeasible scales by focusing them on the time period, tasks and code regions that are most likely to have a bug.

Our evaluation injects synthetic errors into six applications from the NAS Parallel Benchmark (NPB) suite [5] at random time points and in randomly chosen tasks. The errors include delays, hangs in application tasks, interference due to execution of an extra CPU- or memory-intensive thread on an application compute node and message drops and duplication. Our results demonstrate that AutomaDeD correctly identifies the time period that is likely erroneous in 90% of our trials for delays, hangs and message faults and in 70% of our trials for interference faults. Given the correct time period, AutomaDeD's cluster isolation achieves over 80% accuracy for delays and hangs, 40% for message faults and 70% accuracy for interference faults. Given the correct cluster, it isolates the injected transition with 90% accuracy for delays and hangs and 50% accuracy for interference faults.

The rest of the paper is organized as follows. Section 2 presents our overall approach, while Section 3 looks at the details of our application behavior modeling methodology. We describe the analysis performed by AutomaDeD in Section 4 and present our experimental evaluation in Section 5.

2 Approach

As Figure 1 shows, AutomaDeD consists of both on-line and off-line mechanisms. An on-line mechanism gathers data about executions into an SMM database. AutomaDeD's off-line mechanisms then use this data to derive a deeper understanding of the application behavior, particularly when bugs are manifested.

Figure 2. Example of a Semi-Markov Model

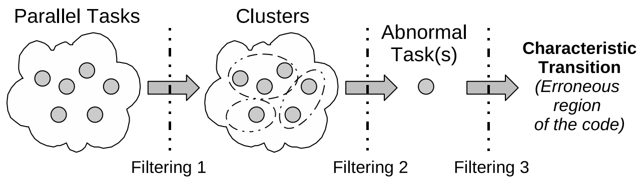

Figure 3. Problem-size reduction with AutomaDeD

2.1 Semi-Markov Models

We model the control flow and timing properties of application tasks in order to debug common anomalies. We track control flow as a sequence of application states, defined as MPI calls (including their arguments and call stack) or the computation interleaved between them. We maintain the amount of time each task spends in each state to capture temporal aspects of the states. Given the expense of maintaining full traces, we model task behavior as a *Semi-Markov Model* (SMM), a finite automaton of task states and transitions where the task spends a random amount of time in each state and randomly selects its next transition with no dependence on its history.

Figure 2 shows a sample SMM with edges labeled by the probability of transitioning from one state to another and the probability distribution of the time preceding the transition. In the above SMM, tasks in state S_3 transition to state S_1 40% of the time and to S_2 the other 60%, with the times that precede the transitions sampled from distributions $F_{3,1}$ and $F_{3,2}$, respectively. We compute the SMM states, transitions and probability distributions from program traces captured on-line by a P^{n}MPI-based wrapper library [20] that intercepts all calls to MPI functions. We use the observed normalized frequency of each transition as its transition probability. Section 3.1 explains how we derive time distributions.

2.2 Overview of Analysis

The SMM abstraction couples the dynamic execution of an application with distinct regions of its code. Thus, we can focus the developer's attention on the tasks and regions of code that are behaving anomalously. Figure 3 shows the several stages in which we accomplish this goal. First, we divide the application's execution into a series of time periods called *phases*. Naturally, applications behave according to a repetitive pattern for periods of time and then their behavior changes, to a different repetitive pattern or some random pattern. We divide the period of time of repetitive behavior into smaller time periods, which we call phases.

Thus, across the phases within one repetitive pattern boundary, we expect the application behavior to be statistically identical. AutomaDeD then computes an SMM for each task within each phase and then clusters the SMMs for each phase. This clustering may partition the tasks based on correct differences between them, such as with master-slave applications that have two correct partitions. Alternatively, it may identify behavioral differences due to a bug. AutomaDeD either compares a task's SMMs from different phases or the task clustering from different phases to determine the phase during which a bug is first manifested. It can also compare SMMs or their clusterings to those from prior, correct executions. If no sample runs are available, AutomaDeD calibrates its bug detection algorithms based on the first phase, which works well in practice because we target rare, hard to find bugs, which manifest themselves after a few iterations of the main processing loop.

Once AutomaDeD identifies a faulty phase, it proceeds to identify the task cluster or individual task in which the bug is first manifested. AutomaDeD compares SMMs or clusters across phases to identify the SMM or cluster that has changed the most from the normal phases. AutomaDeD can again use SMMs or clusters from prior, correct executions or earlier phases of the same execution. AutomaDeD also compares the individual state transitions within the faulty phase to find the first unusual transition, which may identify the error site. Alternatively, the most unusual SMM transition of the faulty task may identify it.

Thus, AutomaDeD iteratively focuses the developer's debugging efforts. First, it identifies the faulty execution phase. Then it finds the faulty task or group of tasks. Finally, it locates the error site. The granularity of this identification is a state in the SMM. Thus, AutomaDeD*does not* identify the root cause of the error and cannot identify the manifestation to a very fine granularity, such as line of code. However, it does significantly reduce the amount of information that must be considered when performing a root cause analysis.

3 SMM Mechanisms

3.1 Creating Time Distributions

We consider two methods for deriving the time probability distributions that explain the time spent by a task in the SMM states. In one, we assume that the time values follow a Gaussian distribution. In the other, we compute a histogram of ranges of the observed time values, instead of assuming a particular distribution.

Assuming Gaussian distribution has several advantages. First, we can easily calculate the parameters of a Gaussian distribution given enough sample points. By using maximum-likelihood estimation, we only need to calculate

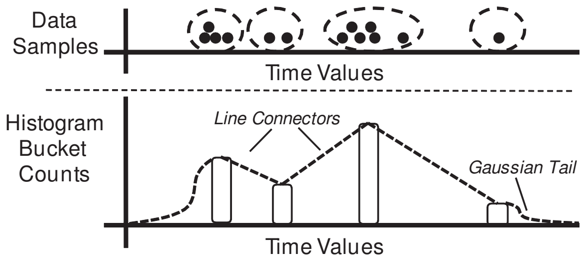

Figure 4. Example of histogram construction

the mean and standard deviation of the data points. Second, it is a well-known distribution with a rich theory. However, a Gaussian distribution is not appropriate for state transitions that have multi-modal or asymmetric behavior. The former can occur when different code within a compute region is executed at different times and the latter occurs when the time that precedes a transition is consistent except for spikes due to system or network interference.

Histograms provide a more detailed fit to the observed data. The basic approach, which Figure 4 shows, divides the observed data points into a number of equal-sized buckets. The probability of a particular bucket is the fraction of data points within it. Since timing data may have outliers orders or magnitude above the median, equal-sized buckets can aggregate most data points into a single bucket, providing poor resolution. We therefore used variable-sized buckets via an online clustering algorithm. We assign each new data point to its own bucket. If the resulting number of buckets rises above a threshold, we merge the two buckets with the closest means. We derive a continuous probability distribution from the discrete histogram by linearly connecting adjoining bucket counts and modeling the regions beyond the smallest and largest buckets using the lower and upper halves of Gaussian distributions, which models the probability of observing new extreme values.

The basic tradeoff between these two distributions is that Gaussians are cheaper (in terms of computation and memory cost to create and to query) and more constrained while histograms are more expensive but very flexible. Evaluating both options provides significant information about the basic tradeoffs of this design parameter, thus illuminating the potential of other statistical models such as mixed-Gaussian distributions and Kernel Density Methods [21].

3.2 Comparing Task SMMs

AutomaDeD detects faulty phases and tasks and performs task clustering by comparing SMMs to each other. We define an SMM distance metric that reflects the differences between the control flow and timing behaviors of their respective tasks. The difference between two SMMs is the sum of the differences in their transition probabilities and transition time distributions.

Given two SMMs A and B, let S_A and S_B be their sets of states, and T_A and T_B be their sets of transitions. Also let $d_{s,i}$ be the state transition probability distribution for state $s \in S_i$, and let $d_{t,i}$ be the time probability distribution for transition $t \in T_i$. The difference between A and B is:

$$Diff(A,B) = \sum_{s \in \mathbf{S}} D(d_{s,A}, d_{s,B}) + \frac{1}{\nu} \sum_{t \in \mathbf{T}} D(d_{t,A}, d_{t,B})$$

where $\mathbf{S} = S_A \cup S_B$, $\mathbf{T} = T_A \cup T_B$, $D(d_{r,A}, d_{r,B})$ is the difference between a pair of probability distributions $d_{r,A}$ and $d_{r,B}$, where r is a state or transition. ν corresponds to a weighting factor defined in Section 3.3 that weighs differences on transitions with consistent timing behavior above those with poor information content. We define the metric $D(d_{r,A}, d_{r,B})$ as:

$$D(d_{r,A}, d_{r,B}) = \begin{cases} L_2(d_{r,A}, d_{r,B}) * \alpha & \text{if } r \in A \text{ and } r \in B \\ 10 & \text{otherwise} \end{cases}$$

$L_2(d_{r,A}, d_{r,B})$ is the L_2 norm between the probability distributions [15], $L_2(d_{r,A}, d_{r,B}) = \int_{-\infty}^{\infty} |d_{r,A}(j) - d_{r,B}(j)|^2 dj$. The integral is over the space of possible events (state transitions or transition times).The parameter α gives greater weight to differences in time distribution with distant means, μ_d and $\mu_{d'}$. For time distributions it is equal to

$$\alpha = 1 + \frac{|\mu_{d_{r,A}} - \mu_{d_{r,B}}|^2}{(\mu_{d_{r,A}} + \mu_{d_{r,B}})/2}$$

and $\alpha = 1$ for state transition distributions.

In most cases $D(d_{r,A}, d_{r,B})$ is below 10 for transitions and states r that appear in both A and B. As such, if r appears in one but not the other, $D(d_{r,A}, d_{r,B})$ was set to 10 to make differences in application control flow more significant than differences in its timing behavior.

3.3 Normalized SMM Comparison

Different SMM transitions will have very different timing properties, with a variety of means, standard deviations and distribution shapes. Differences between SMMs on a transition that has consistent timing and a tightly focused distribution can be very informative. In contrast, if the transition is noisy, the differences are most likely due to system interference. AutomaDeD focuses on the critical differences between two SMMs by looking at the "normal" difference between the SSMs of a sample set and weighting $D(d_{t,A}, d_{t,B})$ accordingly. Thus, given a transition t and a set M of sample SMMs, we define the weighting factor ν as the root-mean-square of D on this transition among the members of M:

$$\nu = \sqrt{\frac{\sum_{A,B \in M, A \neq B} D(d_{t,A}, d_{t,B})^2}{|pairs(t,M)|}}$$

where $|pairs(t,M)|$ is the number of SMM pairs in M that both have transition t. In the absence of sample runs, ν for a given transition in a given phase of the faulty run is computed by summing over SMMs in the run's other phases.

This weighting scheme overcomes a commonly observed effect where certain transitions have multi-modal timing characteristics—very consistent timing behavior within each mode and sudden shifts to a different mode either within a given run or across multiple runs. This may be caused for example by a computation that executes the same set of instructions but takes very different times depending on whether the data is cached or not. For such behavior, the value of ν will be high, thereby weighing down the difference metric D.

3.4 Clustering Tasks' Models

AutomaDeD detects behavioral clusters by using Hierarchical Agglomerative Clustering (HAC) [10] on the SMMs of all application tasks. HAC initially sets each task to be in its own cluster. During each iteration, HAC merges the two most similar clusters into a single cluster, so that it has one cluster less after that iteration. Cluster difference is defined as the smallest difference between any member of one cluster to any member of the other cluster. These steps are repeated until the minimum difference between any pair of clusters is above a given threshold (i.e., no two clusters are similar enough to merge).

HAC requires a threshold that defines the normal difference of similar tasks. AutomaDeD chooses this threshold by having the developer provide the number of clusters that accurately describe the application's expected behavior. For example, a relaxation algorithm with non-periodic boundaries operating on an 2-dimensional grid is best described by a 9 clusters (one for the interior, and one for each side and each corner region). However, it should have a single cluster if the boundaries are periodic. AutomaDeD applies HAC on SMMs of a set of training phases (assumed to have few bugs), identifying the average threshold that produces the desired number of clusters. We use this threshold for subsequent clustering. If sample runs of the application are provided, AutomaDeD trains on phases in these runs. Otherwise, it trains on the given run's first phase, which we assumed is fault-free.

The resulting clustering organizes tasks into behavioral groups that reflect the effect of the bug on the application's behavior. This helps to identify the time and the location when the fault was first manifested.

4 Error Detection Procedure

We describe the procedure that a user employs to isolate a bug using AutomaDeD. Figure 1 shows the complete sequence of steps. *On-line* steps occur when the program executes, while *off-line* steps occur after execution. The next sections describe each step.

4.1 Phases and Epochs

AutomaDeD models the behavior of discrete regions of application execution that the developer identifies via source code markers. The term *phase* denotes a region of execution, such as a time step, that repeats multiple times. Phases are grouped into `phase sets`, where all phases in a set are assumed to behave similarly to each other. For example, adaptive mesh refinement applications periodically re-partition their work and meshes. Thus, individual iterations may be identified as phases while iterations between adjacent re-partitionings may be grouped into a set. Developers annotate phases and phase sets in their code by adding calls to `MPI_Pcontrol`, a special function call that is intercepted by our wrapper library.

4.2 Faulty Phase Detection

AutomaDeD detects the phase during which a fault was first manifested using one of two algorithms, depending on how it effects application behavior. Currently the user of AutomaDeD must try both faulty phase selection mechanisms. We leave automation of this selection to future work.

If the effects are temporary (e.g., temporary delay due to unusual erroneous control flow), AutomaDeD searches for the phase that differs from all other phases. If AutomaDeD has a set of sample runs, it compares each phase to its counterparts in those runs. It can either compare each task's SMM directly to its sample counterpart or it may compare each phase's clustering to the clustering of its counterpart phase. For the former we use the SMM difference metric from Section 3.2, with the difference between two phases defined as the squared sum of the differences between their respective task SMMs. For the latter we use the Mirkin difference metric [17], which is the fraction of task pairs that are grouped differently in the two clusterings, (i.e., tasks T_1 and T_2 are in the same cluster in one clustering and not in the same cluster in the second, or vice-versa). Then for each phase we compute a "deviation score", which is the sum of the squared distances from this phase in the faulty run to the same phase in the sample runs. We identify the phase with the highest deviation score as faulty. If no sample runs are provided, AutomaDeD compares each phase to all others within the faulty run using either of the above metrics to compute each phase's deviation score. We identify the phase that differs most from the others as faulty. When sample runs are provided, ν weighting terms are computed from the SMMs of these runs. When they are not provided, the ν used for each phase's comparisons is computed from the other phases in the faulty run.

If the effects are permanent (e.g., a runaway thread that interferes with the application), AutomaDeD identifies the phase when application behavior shifted. If AutomaDeD has sample runs, it computes deviation scores as above but then uses k-Means Clustering [10] to divide the phases into two clusters: those that are similar to the sample runs (low deviation) and those that are different (high deviation). We identify the earliest phase in the high deviation cluster as faulty. Without sample runs, AutomaDeD identifies the pair of adjacent phases that are most different according the SMM or clustering difference metrics. The later phase in this pair is judged to be faulty.

4.3 Pinpointing Faulty Task(s) and Error Sites Using SMM Analysis

AutomaDeD provides two complementary mechanisms to identify the faulty task(s) and the error site. We describe the first mechanism, which compares SMMs and clusterings, here. We discuss the second, which is based on individual transitions, in Section 4.4. Successful identification of the faulty cluster greatly simplifies determining the root cause. Cluster isolation is particularly helpful when the manifestation of a bug results in a cluster with a single task.

AutomaDeD clusters the tasks of the faulty phase and then identifies the most unusual cluster by computing its deviation from the other clusters. If we have a set of sample runs, we compare each cluster to them using the SMM or clustering difference metric (comparison is focused on the phase identified as faulty). The SMM cluster difference is simply the sum of the squared SMM differences between the SMMs of member tasks in the faulty phase and their SMMs in the same phase of a given sample run, divided by the number of tasks. The clustering difference metric is a variant of the Mirkin difference where the deviation from sample phase clustering $C' = \{c'_1, ...c'_n\}$ of test cluster c is the fraction of its member task pairs that appear in different clusters in C'. Each cluster's overall deviation score is then the squared sum of its differences with respect all the sample runs. If no sample runs are provided each cluster is compared as above but to the other phases of the faulty run instead of the same phase of the sample runs.

We also locate the error site based on task clustering when we identify the *characteristic transition* (CT), the transition that most distinguishes the faulty cluster from the other clusters. Since bugs can cause these behavioral differences, CTs direct developers to the root cause.

For SMMs A and B, we define $CT(A, B)$ as (t, χ) where t is transition that most contributes to the dissimilarity metric $Diff(A, B)$ and χ is the magnitude of this contribution. Given a cluster $c = \{M_1, M_2, ..., M_n\}$, we compute the cluster's CT by evaluating $CT(M_i, M'_j)$ for each pair ($M_i \in c$, $M'_j \notin c$). The CT of c is then the transition that is the CT of the most SMM pairs. If this selects more than one transition, the CT is transition with the largest average contribution magnitude. Since this method does not always produce the correct faulty transition as the top CT, AutomaDeD can also present the top several choices to the developer for closer examination.

4.4 Detection Using Transition Analysis

Our SMM-based cluster and transition isolation methods are too coarse if the effects of the bug propagate to the entire application and will fail to identify the first task(s) and transitions that the bug impacted. We can overcome this difficulty by observing individual state transitions, looking for the first that takes an unusual amount of time compared to the transition behavior seen in sample runs or earlier phases.

If the faulty effects are temporary, AutomaDeD computes the typical behavior of each SMM transition as a probability distribution (Gaussian or Histogram) of its observed times in the sample runs or first phase, after discarding the top and bottom 1% of the times. AutomaDeD uses these distributions to compute the probability of observing the time preceding each transition of the faulty phase. We then identify low probability transitions through K-Means clustering with $K = 2$, using the log of the probability to improve sensitivity to low values. We select the earliest low probability transition as the CT, which also identifies the faulty task. AutomaDeD can also present later low probability transitions on other tasks in case the starting times of the transitions do not correctly identify the CT.

If the faulty effects are permanent, AutomaDeD looks for a sudden change from one type of application behavior to another. Specifically, it scans each transition t in each task SMM M to locate the largest increase in $\theta = stdDev(t) * \overline{\nu}$, where $stdDev(t)$ is the standard deviation in the observed times preceding t. When sample runs are provided, $\overline{\nu} = \frac{1}{\nu}$, where ν is the noise weighting factor discussed in Section 3.3. Otherwise, $\overline{\nu} = stdDev(t)$, which is another way to reduce the algorithm's sensitivity to outliers.

θ measures the variation of the transition, which increases significantly when its behavior changes, as its prior behavior does not predict its new behavior well. AutomaDeD selects the transition that provides the best balance between occurring before other transitions and having a high θ. This is done by comparing transitions t and t' using to the following relation:

$$(t_{ts}, \theta) \succ (t'_{ts}, \theta) \equiv \begin{cases} \theta * (1 + t'_{ts} - t_{ts}) > \theta' & \text{if } t_{ts} < t'_{ts} \\ \theta' * (1 + t_{ts} - t'_{ts}) > \theta & \text{if } t'_{ts} < t_{ts} \end{cases}$$

where t_{ts} and t'_{ts} are the timestamps of t and t'. Thus, we consider t a better choice (ordered larger) than t' if either it has an earlier timestamp and θ is larger than θ' after being adjusted by a factor that proportionally compensates for the difference in their timestamps or it has a later timestamp and θ is larger despite θ' being inflated by the same factor.

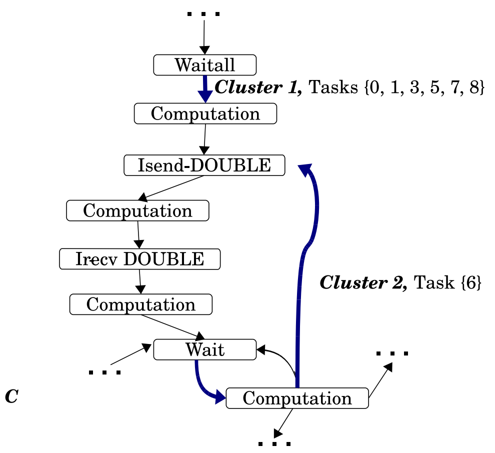

Figure 5. Output format of AutomaDeD after the debugging process is completed.

4.5 Visualization of Results

AutomaDeD presents the cluster and transition isolation results through the clustered SMMs of the faulty phase, focusing on the faulty cluster and the CT. Figure 5 shows an example of the output for a 9 task NAS benchmark BT when a 10 second delay was injected into task 6 before execution of the selected `MPI_Isend` (we show only a portion of the SMM). Bold edges indicate the CTs; the clusters appear as their labels. The cluster associated with the edge (Computation, Isend-DOUBLE) corresponds to the faulty cluster.

5 Experimental Evaluation

5.1 Fault Injection Types

We empirically evaluate the effectiveness of AutomaDeD by injecting synthetic faults into six applications in the NAS Parallel Benchmark suite: BT, CG, FT, MG, LU and SP [5]. We omitted EP because it performs almost no MPI communication and IS because it uses MPI in only a few locations in the code, making MPI-based state demarcation inappropriate. Our fault injector, built on top of P^{N}MPI, dynamically injects a wide array of software faults at random MPI calls during MPI application runs. It supports three main classes of faults:

- Local livelock/deadlock or transient stall; emulated via a finite loop of 1, 5 or 10 seconds (`FIN_LOOP`) or an infinite loop (`INF_LOOP`)
- MPI message loss and duplication; emulated by dropping (`DROP_MESG`) or repeating (`REP_MESG`) a single

MPI message,

- Extra CPU- or Memory-intensive thread; emulated by starting up a thread with a perpetual-increment loop (CPU_THR) or a loop that randomly reads from/writes to a 1GB region of memory (MEM_THR), that interfere with the remainder of the application's execution.

Our experiments ran each benchmark with input size A and 16 tasks. We executed all tasks on four-socket, quad-core nodes (the Hera cluster at LLNL), with 2.3Ghz Opteron processors and 32GB RAM per node. We injected each fault type into a random task and MPI operation type (Blocking and Non-Blocking Sends and Receives, All-to-Alls, etc.), ensuring that over the entire experiment, each task and MPI operation type was injected with each fault type. For each case, we performed at least 10 random injection runs, totaling approximately 2,000 injection experiments per application. In each run we injected a single fault into a random instance of the target operation type on a random task. The execution of each application was partitioned into approximately 5 phases; the exact number depended on the application's original iteration count.

5.2 Results of Debugging Faults

We evaluate the accuracy of AutomaDeD in identifying the following aspects of the injected fault:

- The phase with the injected fault (faulty phase)
- The cluster that contains the task with the injected fault (cluster isolation)
- The error site of the injected fault (transition isolation)

We evaluate AutomaDeD with and without sample runs. Using sample runs corresponds to when the developer can execute an application multiple times to establish its normal behavior before analyzing a given faulty run. We evaluate two types of sample runs. For each application `A` the `FaultFree(A)` set consists of 20 runs with no injected faults, which models an ideal set of sample runs. The `Fault10(A, F)` set includes `FaultFree(A)` as well as 2 additional runs of `A` in which fault `F` was injected. This set models the more common case where application runs are affected by an infrequent non-deterministic bug that affects a certain fraction of runs (in this case ∼10%). Our experiments that do not use sample runs, denoted `NoSample`, omit any runs in which faults were injected during the first phase in order to ensure a more informative evaluation. We also omit such runs when analyzing CPU_THR and MEM_THR faults, regardless of whether or not sample runs are provided, since they provide no information about the application's behavior before the fault.

5.2.1 Detection of the Faulty Phase

We begin by evaluating AutomaDeD's ability to detect the phase in which the fault was injected. If AutomaDeD does not have sample runs, the algorithm identifies the phase that is most different from the others using either the cluster-based metric or the individual task SMM-based metric. If it has sample runs, AutomaDeD use one of these metrics to determine the phase that is most different from its counterpart in those sample runs.

Figure 6 shows the average accuracy over all applications of faulty phase detection. All of our graphs show the runs on the Y-axis in which AutomaDeD identifies the phase, cluster or transition relevant to the injected fault. The data series correspond to using the two metrics with each sample run configuration (`FaultFree`, `Fault10` and `NoSample`) and the different distribution methods used for the times preceding transitions (`Gaussian` and `Histogram`).

We observe that the SMM-based metric detects faulty phases more accurately than the cluster-based one, with detection accuracy over 90% for most fault types. However, the cluster-based metric better detects CPU_THR and MEM_THR when sample runs are available. In general, sample runs significantly improve faulty phase detection accuracy, with `FaultFree` and `Fault10` generally exceeding `NoSample` by 20%-30%. The difference is even larger for CPU_THR and MEM_THR. Further, `FaultFree` and `Fault10` sample runs provide similar accuracy, which suggests that moderate noise levels do not impact the SMM representation and AutomaDeD's analyses significantly. Finally, SMMs based on `Histograms` produce consistently more accurate (by several percent) phase detection results than those based on `Gaussian` probability distributions because they are less sensitive to noise such as outliers. We observe similar trends for cluster and transition isolation.

Figure 7 shows faulty phase detection accuracy on a per-application basis, focusing on SMMs that use `Histogram`. The data shows that detection accuracy depends strongly on the application. Further, the SMM-based metric has poorer accuracy with CPU_THR and MEM_THR primarily due to its poor results on MG, BT and SP, while it provides higher accuracy for FT than does the cluster-based metric. The SMM-based metric is more accurate for other faults because it performs more consistently across the applications. Finally, sample runs are essential for the cluster-based metric while the SMM-based metric still provides reasonable accuracy on the other fault types without them.

5.2.2 Cluster Isolation

Once AutomaDeD identifies the faulty phase, its cluster isolation can help locate the root cause of the bug by showing the cluster that contains the task where the bug was injected.

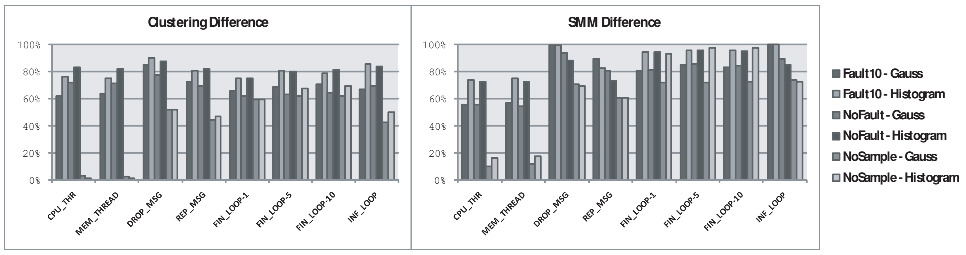

Figure 6. Average faulty phase detection accuracy

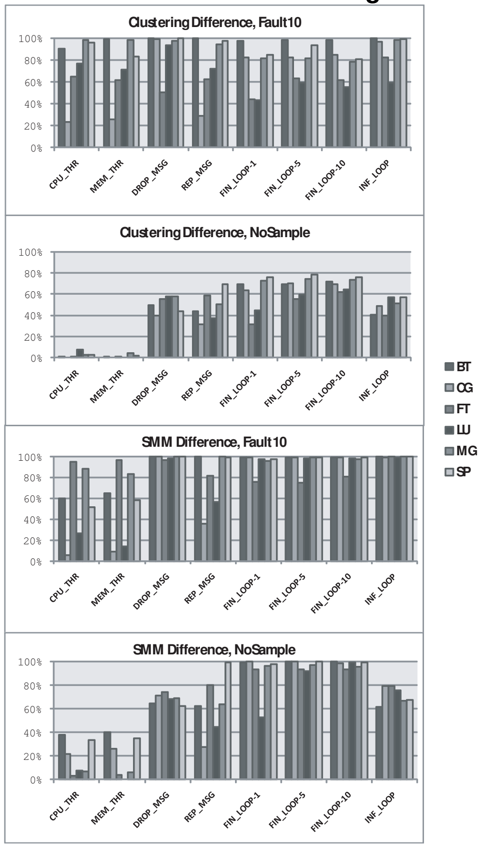

Figure 7. Faulty phase accuracy per application

AutomaDeD again uses the SMM-based and cluster-based metrics to perform cluster isolation. Alternatively, it can examine the individual transitions within the faulty phase to identify those that are unlikely given the probability distribution on the transition. Our evaluation measures the accuracy of AutomaDeD's cluster isolation separately from that of its faulty phase detection by always applying the techniques to the faulty phase, that is we assume the phase detection was accurate.

Figure 8 shows the accuracy of AutomaDeD's cluster isolation on a per-application basis, focusing on SMMs that use `Histogram`. Cluster isolation using the cluster-based metric has poor accuracy for nearly all applications and fault types. The other options produce significantly better results. The abnormal transition method without sample runs and the SMM-based metric with sample runs provide the best accuracy for `CPU_THR` and `MEM_THR`, with near perfect results on half the applications. The abnormal transition method achieves high accuracy for the `FIN_LOOP` and `INF_LOOP` using sample runs.

In general, the accuracy of cluster isolation varies widely across the applications for the same fault type since the faults can propagate themselves quickly from one task to another. Thus, some task(s) other than the faulty task may exhibit behavior the most divergent from its normal activity, which can cause the cluster-based and SMM-based metrics to mis-identify them as the source of the fault. While task behavior does not confuse the abnormal transition method, it can perform poorly due to the relatively coarse granularity of SMM transitions. As such, a fault may propagate from a transition with a later starting timestamp to one that began earlier, causing the wrong transition to be identified as the fault's first manifestation. We could reduce this effect by breaking long states into smaller ones, which will improve their precision.

Figure 9 shows the percentage of runs (using the `Fault10` sample run configuration) in which the faulty task cluster consists of only one faulty task. Precisely identifying the faulty makes significantly easier for developer to identify the bug since narrows it down to a single task's control and data flow. AutomaDeD fully isolates the faulty task in more than 90% of the cases for `CPU_THR`, `MEM_THR`, `DROP_MESG` and `REP_MESG` and 70% for `FIN_LOOP` and `INF_LOOP`. In contrast to prior results, using `Gaussian` distributions for the times preceding transitions provides greater accuracy because they are more sensitive to outliers, which suggests that both probability distributions should be used in practice.

5.2.3 Transition Isolation

AutomaDeD uses two algorithms for transition isolation. First, it compares the SMMs of the faulty cluster to those of other clusters and selects the transitions most responsible for the differences. Alternatively, it selects the earliest abnormal transition within the faulty cluster. Since our goal is to focus debugging efforts, we consider how frequently the faulty transition is the top choice or one of the top five choices of these methods. Figure 10 shows the results, with the clustering-based algorithm on the left and the transition-based algorithm on the right.

The clustering-based algorithm consistently ($\geq$ 90% of

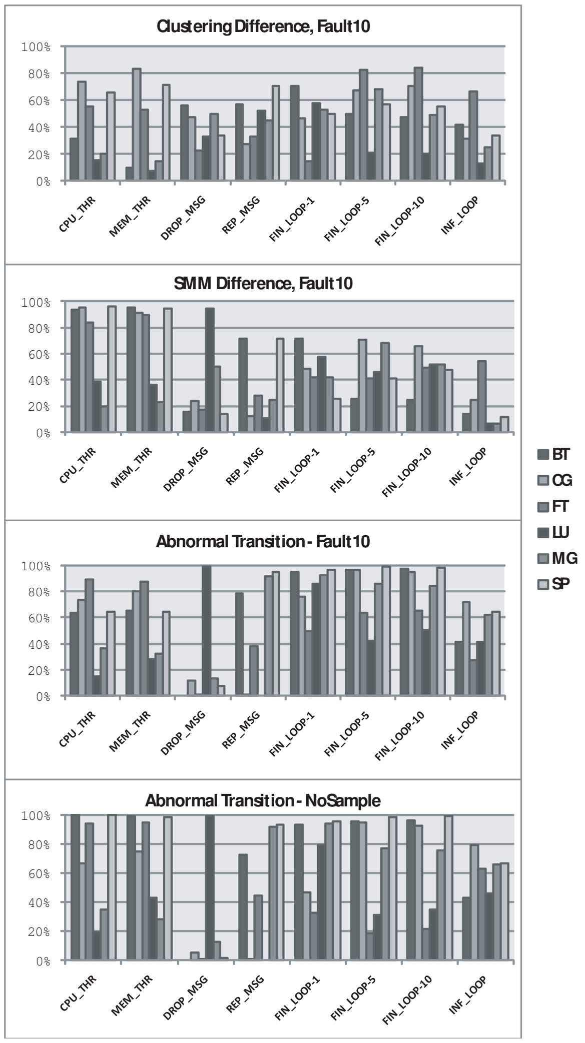

Figure 8. Cluster isolation accuracy per application

the time) includes the faulty transition in its top five choices for FIN_LOOP and INF_LOOP. The transition-based algorithm is less consistent across applications but when it succeeds, it usually does so with its first selection. Both methods exhibit low accuracy for DROP_MESG and REP_MESG faults because their effects manifest long after the fault is injected. They also perform relatively poorly with CPU_THR and MEM_THR because these faults cause sudden behavioral changes that resemble ordinary outlier transitions.

5.3 Case Study: MVAPICH Bug

We illustrate the utility of AutomaDeD via a case study of applying it to a real bug in the MVAPICH-0.9.9 MPI implementation [19]. The bug occurs in its MPI task launcher, mpirun, which sometimes fails to clean up after an application, leaving processes to run concurrently with subsequent jobs. We evaluated AutomaDeD on this bug, which is similar in effect to our CPU- and Memory-intensive thread faults, by executing a 16- or 64-task run of as the application being debugged while simultaneously executing a 16-task run of either LU, MG or SP on the same set of nodes as the previous runaway tasks). These experiments cover the cases where runaway tasks interfere with either all or a subset of the application's tasks.

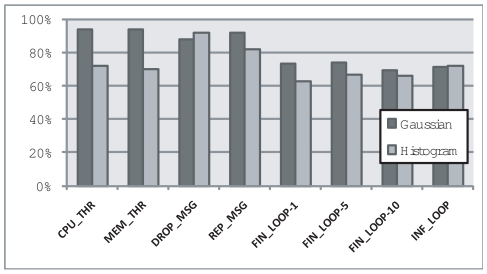

Figure 9. Isolation of a singleton cluster

We provided AutomaDeD with a set of five sample runs of BT with no interference. Figure 11 presents average the SMM-based metric that AutomaDeD determines for each phase of three runs where BT ran concurrently with either LU, MG and SP (one set for 16-task and another for 64-task BT runs) as well as the average score for the five no-interference runs. The sets of sample runs used to compute each no-interference run's deviation scores excluded the run itself. The deviation scores of all no-interference phases were consistently low. In contrast, the scores of the initial phases of the three interference runs show high deviation scores, identifying the exact region of time when the shorter runs of LU, MG and SP overlapped with the execution of BT. Further, AutomaDeD clearly shows that the interferfence run of MG in one 16-task experiment began after the first phase of BT, since the deviation score starts at the baseline level, rises for three phases and then drops to the baseline.

AutomaDeD significantly aids debugging. First, it clearly identifies the performance anomaly, which might not have been noticed for a long time or blamed on extraneous factors such as network load or choice of input. Second, AutomaDeD determines when the interference occurs, which facilitates detection of the interference tasks from system logs or other methods. Although AutomaDeD can often identify the tasks most affected by the fault, it did not isolate those tasks in this case since BT is tightly coupled, which leads to the interference tasks impacting all of BT's tasks even with 64-task runs.

6 Prior Work

Traditional debugging techniques, including sequential debuggers such as gdb and "printf debugging," require the user to identify coding errors and to trace their origins manually. Traditional parallel debuggers, such as TotalView [23] and DDT [2], are similar, although these tools must control multiple processes concurrently and aggregate

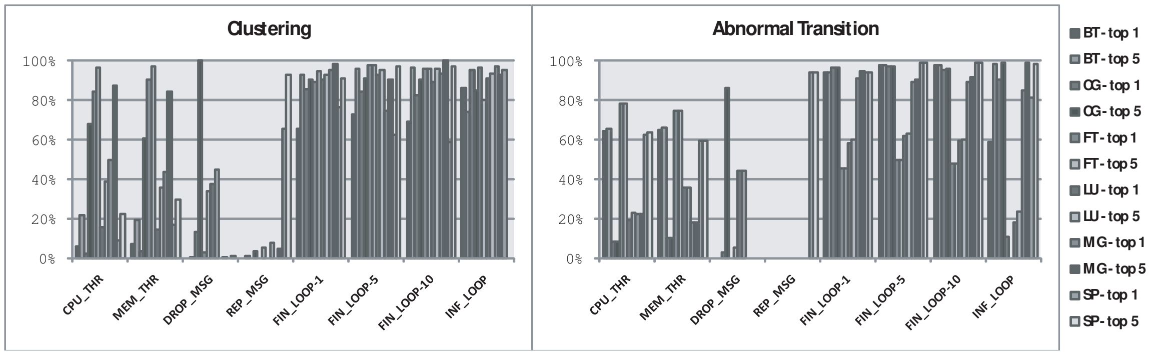

Figure 10. Transition isolation accuracy per application

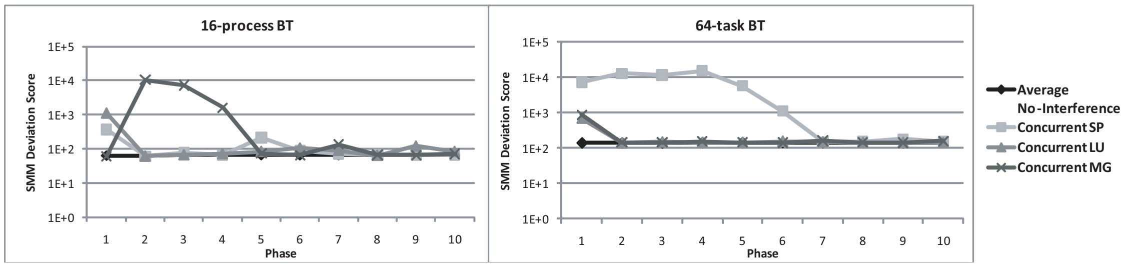

Figure 11. Phase deviation scores of MVAPICH bug use-case

the distributed state. They provide convenient interfaces to that state but the process of identifying errors remains manual. Overall, traditional debugging techniques require a significant amount of user experience, intuition, and time, and thus are not practical for large, complex parallel applications.

Differential debugging provides a semi-automated approach to the analysis and understanding of programming errors by comparing executions dynamically [1, 24]. Recent research has focused on developing statistical techniques to pinpoint the root cause of correctness problems automatically [3, 6, 8, 9]. While both approaches hold significant promise, they require extensive runtime execution data often from multiple runs or extensions that guide the analyses to the significant differences between the processes in a single run. As a result, most techniques have not yet been applied to large scale runs of parallel applications. AutomaDeD complements these techniques by providing mechanisms that relate the state across the individual processes and group ones with similar behavior into task equivalence classes.

Our previous work developed the Stack Trace Analysis Tool (STAT) [4, 11], which provides scalable task equivalence class detection based on the functions that the processes execute. Specifically, STAT gathers stack traces across tasks and over time and merges the traces into a call graph prefix tree, from which it identifies task equivalence classes. STAT's stack trace analysis is useful for diagnosing certain classes of errors. STAT can quickly identify when a small subset of tasks has diverged from the rest of the application. However, these classes can group processes when their behaviors are distinct. For example, two processes can exhibit the same stack trace despite having very divergent timing characteristics and the timing characteristic of one of the processes indicates that it is erroneous. Hence, AutomaDeD considers extensions over function-based equivalence classes through the use of SMMs with state transition probabilities and timing distributions. Further, distinct from prior work, we also drill down and identify the code region that causes the divergence, thus providing a guidance to the developer to look for errors.

Liu *et al.* present D^{3}S, which provides users with a model to write predicates on properties that they wish to check at runtime [13]. The system provides a scalable means to verify the properties, and tolerates failures of the checking and the checked machines. The P2 monitor [22] is similar in spirit to D^{3}S—differences lie in the level of automation in collecting distributed state and support for legacy applications. All of this prior work differs from ours in that they focus on detecting when a property gets violated, with the property being well understood and specified by the user prior to execution while we do not require the user to provide such knowledge.

We share similar goals to those of the work by Mirgorodskiy *et al.* [16], namely, locating the causes of anomalies in parallel programs. Their model looks at the traces of function calls and exits and uses a distance metric to identify the trace that is most different from other traces. Subsequently, they identify the function that most contributes to the suspect score for the outlier trace in order to pinpoint the likely source of the problem. Despite these similarities, their work addresses a subset of the anomalies that we do in

AutomaDeD since they assume all processes have identical behavior and they do not consider timing anomalies.

7 Conclusion

Large-scale application debugging is very challenging because of the vast amount of information developers must consider to identify a bug's root cause. AutomaDeD focuses debugging efforts on the time period, tasks and code region where the bug is first manifested. Thus, it significantly improves developer debugging productivity by reducing the amount of information that must be considered even as the application is scaled to large task counts. This paper describes the fundamental approach and design of AutomaDeD and establishes it as a valuable addition to the developer's toolkit. Our results demonstrate that AutomaDeD is very accurate for key debugging tasks. In particular, it correctly identifies the faulty phase in 90% of our trials for delays, hangs and message faults and in 70% of our trials for interference faults. Given the faulty phase, AutomaDeD's accurately identifies a small task set (often a single task) in which the bug occurred for over 80% of delays and hangs, over 40% for message faults and over 70% for interference faults. Given the faulty cluster, AutomaDeD identifies the error site with 90% accuracy for delays and hangs and 50% accuracy for interference faults.

While this paper demonstrates the utility of our approach, a key component of our ongoing work is to make these ideas work at large scale. This includes developing more efficient algorithms for our basic mechanisms such as histograms and SMM comparisons, as well as scalable methods to cluster SMMs on-line across millions of tasks. While this work will leverage the algorithms presented here, it will involve the development on novel statistical modeling techniques that can scale to millions of tasks. Another important area will be extending AutomaDeD to model a richer space of behaviors, including analyzing behavioral metrics other than control flow and time as well as modeling more complex applications. This work will enable AutomaDeD to become a valuable debugging tool for developers of large scale applications that will make them significantly more productive even as their applications scale to ever more tasks.

References

[1] Abramson, D., Foster, I., Michalakes, J., and Socič R. Relative Debugging: A New Methodology for Debugging Scientific Appl ications. *Communications of the ACM 39*, 11 (1996), 69–77.

[2] Allinea Software. Allinea DDT the Distributed Debugging Tool.

[3] Andrzejewski, D., Mulhern, A., Liblit, B., and Zhu, X. Statistical Debugging Using Latent Topic Models. In *18th European Conference on Machine Learning* (Sept. 17–21 2007), S. Matwin and D. Mladenic, Eds.

[4] Arnold, D. C., Ahn, D. H., de Supinski, B. R., Lee, G. L., Miller, B. P., and Schulz, M. Stack Trace Analysis for Large Scale Debugging. In *The International Parallel and Distributed Processing Symposium* (2007).

[5] Bailey, D., Barton, J., Lasinski, T., and Simon, H. The NAS Parallel Benchmarks. RNR-91-002, NASA Ames Research Center, Aug. 1991.

[6] Chilimbi, T., Liblit, B., Mehra, K., Nori, A., and Vaswani, K. Holmes: Effective Statistical Debugging via Efficient Path Profiling. In *31st International Conference on Software Engineering (ICSE)* (May 2009).

[7] Feldman, M. Lawrence Livermore Prepares for 20 Petaflop Blue Gene/Q. In *HPCwire* (Feb. 2009).

[8] Gao, Q., Qin, F., and Panda, D. K. DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements. In *ACM/IEEE Supercomputing Conference (SC)* (2007), ACM, pp. 1–12.

[9] Hangal, S., and Lam, M. S. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE '02: Proceedings of the 24th International Conference on Sof tware Engineering* (2002), ACM, pp. 291–301.

[10] Jain, A. K., Murty, M. N., and Flynn, P. J. Data Clustering: A Review. *ACM Computing Surveys 31*, 3 (1999), 264–323.

[11] Lee, G. L., Ahn, D. H., Arnold, D. C., de Supinski, B. R., Miller, B. P., and Schulz, M. Benchmarking the Stack Trace Analysis Tool for BlueGene/L. In *International Conference on Parallel Computing: Architectures, Algorithms and Applications (ParCo)* (2007).

[12] Lindekugel, K., DiGirolamo, A., and Stanzione, D. Architecture for an Offline Parallel Debugger. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (Dec 2008), pp. 227–235.

[13] Liu, X., Guo, Z., Wang, X., and Chen, F. D^{3}S: Debugging Deployed Distributed Systems. In *USENIX Symposium on Networked System Design and Implementation (NSDI)* (2008), pp. 423–437.

[14] Lourenço, J., and Cunha, J. C. Fiddle: A Flexible Distributed Debugging Architecture. In *International Conference on Computational Science (ICCS)-Part II* (2001), Springer-Verlag, pp. 821–830.

[15] Manning, C. D., and Schtze, H. *Foundations of Statistical Natural Language Processing*. Cambridge, Mass: MIT Press, 1999.

[16] Mirgorodskiy, A., Maruyama, N., and Miller, B. Problem Diagnosis in Large-Scale Computing Environments. In *ACM/IEEE Supercomputing Conference (SC)* (2006), pp. 11–23.

[17] Mirkin, B. G. *Mathematical Classification and Clustering*. Kluwer Academic Press, 1996.

[18] MPIPlugin. MPI Plugin for KDevelop. `http://sourceforge.net/projects/mpiplugin/`.

[19] MVAPICH Project. MVAPICH Discussion List. `http://mail.cse.ohio-state.edu/pipermail/mvapich-discuss/2007-July/000932.html`.

[20] Schulz, M., and de Supinski, B. R. PNMPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In *ACM/IEEE Supercomputing Conference (SC)* (2007), ACM, pp. 1–10.

[21] Silverman, B. W. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall, 1986.

[22] Singh, A., Maniatis, P., Roscoe, T., and Druschel, P. Using Queries for Distributed Monitoring and Forensics. *Operating Systems Review 40*, 4 (2006), 389–402.

[23] TotalView Technologies. TotalView Debugger. `http://www.totalviewtech.com/productsTV.htm`.

[24] Watson, G., and Abramson, D. Relative Debugging for Data-Parallel Programs: A ZPL Case Study. *IEEE Concurrency 8*, 4 (2000), 42–52.