Large Scale Debugging of Parallel Tasks with AutomaDeD [*]

Ignacio Laguna[2], Todd Gamblin[1], Bronis R. de Supinski[1], Saurabh Bagchi[2], Greg Bronevetsky[1], Dong H. Anh[1], Martin Schulz[1], Barry Rountree[1]

[1]Lawrence Livermore National Laboratory, Computation Directorate, Livermore, CA 94550
{tgamblin, bronevetsky, ahn1, bronis, schulzm, rountree4}@llnl.gov

[2]Purdue University, School of Electrical and Computer Engineering, West Lafayette, IN 47907
{ilaguna, sbagchi}@purdue.edu

ABSTRACT

Developing correct HPC applications continues to be a challenge as the number of cores increases in today's largest systems. Most existing debugging techniques perform poorly at large scales and do not automatically locate the parts of the parallel application in which the error occurs. The overhead of collecting large amounts of runtime information and an absence of scalable error detection algorithms generally cause poor scalability. In this work, we present novel, highly efficient techniques that facilitate the process of debugging large scale parallel applications. Our approach extends our previous work, AutomaDeD, in three major areas to isolate anomalous tasks in a scalable manner: (i) we efficiently compare elements of graph models (used in AutomaDeD to model parallel tasks) using pre-computed lookup-tables and by pointer comparison; (ii) we compress per-task graph models before the error detection analysis so that comparison between models involves many fewer elements; (iii) we use scalable sampling-based clustering and nearest-neighbor techniques to isolate abnormal tasks when bugs and performance anomalies are manifested. Our evaluation with fault injections shows that AutomaDeD scales well to thousands of tasks and that it can find anomalous tasks in under 5 seconds in an online manner.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Reliability, Performance

[*]Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344. (LLNL-CONF-486911)

1. INTRODUCTION

As today's High Performance Computing (HPC) applications increase in complexity, debugging errors, performance anomalies, and unexpected behavior in these applications becomes excessively difficult. A single bug in an HPC application often affects multiple processes, so growing application scales leads to an effect on many processes. Most existing debugging techniques as implemented in tools like gdb [13], TotalView [25], or DDT [5] do not automate the debugging process: developers must manually locate errors and backtrack through interactions across processes to locate the root cause. Clearly, this approach is infeasible for large-scale parallel applications.

We previously developed AutomaDeD [8], a tool that detects errors based on runtime information of control paths that the parallel application follows and the times spent in each control block. AutomaDeD suggests possible root causes of detected errors by pinpointing, in a probabilistic rank-ordered manner, the erroneous process and the code region in which the error arose. Intuitively, the erroneous tasks often form a small minority of the full set of tasks. Hence, they are outliers when we cluster the tasks, based on their features related to control flow and timing. Further, in the time dimension, the executions in the first few iterations are more likely to be correct than in later iterations, which we also leverage to determine correct or erroneous labels.

Almost all existing parallel debugging tools (including AutomaDeD [8]) *fail to scale to the process counts of today's state-of-the-art systems.* Three main factors impede scalability. First, the tools include a centralized component that performs the data analysis. Thus, tools must stream behavioral information from all the processes to this central component so that it can process the information to determine the error and, possibly, its location. Second, the tools require huge amounts of data. While many tools optimize the monitoring part quite well, the cost of shipping all information to the analysis engine and the cost of analyzing the full volume of data remains. While tools such as STAT [15] reduce the data volume that the central component must handle, they still must process the full data in their communication structure. Third, the data structures used to maintain the information are not completely optimized for the operations that need to be performed for error detection and localization, such as comparison of information from processes that belong to the same equivalence class. Small differences in the cost of one operation, though insignificant for hundreds of processes, become significant at larger scales.

Due to scaling limitations, many existing techniques [11, 12, 21] collect information at runtime and perform analysis offline, decoupled from the main application execution. This approach may allow bug diagnosis only after a long execution in the erroneous mode. This reduces application throughput and wastes computational resources.

We aim to change the debugging scenario fundamentally. We propose techniques to perform error detection and diagnosis online. In this work, we introduce novel scalable mechanisms that allow AutomaDeD to execute online in large-scale systems[1]. To achieve this goal, we introduce three major design and implementation innovations:

Efficient edge comparison allows AutomaDeD to compare per-process graph elements efficiently. Following the model of our baseline implementation, AutomaDeD represents each process as a graph in which nodes are MPI calls or computation blocks between such calls. Edges in the graph have a transition probability and a time distribution. We compare edges by representing state information in the graphs using pointers instead of character strings, so that AutomaDeD can perform state comparisons with a few machine instructions (distinct from the baseline, which used character strings to represent application states). We perform optimal comparison of per-edge probability distributions using a lookup table instead of computing an integral.

Graph compression reduces the time to compare the behaviors of processes by merging edge chains, since the difference between two graphs is the sum of the differences between their edges. This mechanism reduces data dimensionality so that AutomaDeD can focus first on finding outliers (i.e, abnormal processes) using fewer dimensions, and later focus on the dimensions that a fault most affects (e.g., by using just that graph region). Our compression preserves the basic control structure of the program so that the analysis provides actionable results.

Scalable outlier detection uses distributed sampling techniques to find the erroneous processes among many parallel ones with low overhead. We use scalable clustering [10] and a novel nearest neighbor technique to find outliers efficiently.

Careful integration of the three techniques eliminates any centralized element in our solution, thus making AutomaDeD scalable to larger and larger system sizes. This benefit requires that we must carefully sample AutomaDeD's data so our input is representative of all processes and we do not miss sharp discontinuities in process behavior.

Our experiments on a Linux cluster with thousands of cores show that AutomaDeD scales to thousands of processes and that it can isolate erroneous tasks in a few seconds. We demonstrate its error-detection capabilities through fault injections in the NAS Parallel Benchmarks and its scalability on up to 5,000 AMG2006 [14] processes. AutomaDeD performs the entire error-detection analysis (i.e., abnormal process and erroneous code region isolation) in under 5 seconds.

The remainder of this paper is structured as follows. In Section 2 we present background material on our baseline. In Section 3 we detail the design of the three novel aspects in AutomaDeD which improves the operational flow of the baseline approach. In Section 4 we present detailed experiments that demonstrate the effectiveness and performance of our scalable error detection techniques.

[1]We refer to our prior work [8] as baseline instead of AutomaDeD to distinguish it from our current work.

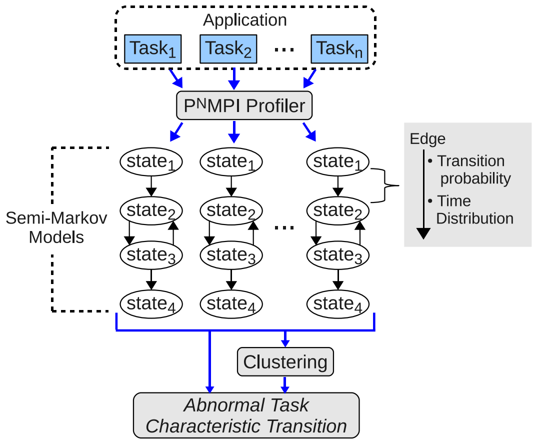


Figure 1: Main blocks in the design of AutomaDeD

2. BACKGROUND

In our previous work, we presented AutomaDeD [8], a tool that allows developers to focus debugging effort on the erroneous period of time, parallel task and code region that are affected by faults in parallel applications. Figure 1 shows the main building blocks in AutomaDeD. Control flow and timing information of each task are modeled using *semi-Markov models* (SMM). Through non-intrusive runtime monitoring, AutomaDeD creates an SMM for each task in the application. States in an SMM represent *communication* code regions, i.e., MPI communication routines, and *computation* code regions, i.e., code executed between two MPI communication routines. A state consists of call stack information such as the module name and offset of the function calls that are currently active in the MPI process. This call stack information is obtained by instrumenting the application dynamically using the DynInst tool [2] (or by using the backtrace API from the GNU C library [3] when DynInst is not available) at the beginning and at the end of MPI routines with a P^{N}MPI [24] profiler. Every edge is a tuple (s, d), representing a transition between source state s and destination state d. Two *attributes* are assigned to each edge: a transition probability that captures the frequency of occurrence of the transition, and a time probability distribution that allows us to model the time spent in the source state conditioned on the destination state.

The application's execution is divided into a series of time periods called *phases* in which the application repeatedly exhibits the same execution pattern. First, AutomaDeD performs clustering of SMMs to find the phase in which the error is first manifested. We detect this phase by looking for sharp deviations in the clustering of SMMs from prior phases. Then, AutomaDeD performs clustering of the SMMs of the different tasks in the erroneous phase to determine the erroneous task(s). The natural number of clusters in an application can be provided by the developer (in cases, the distinct functionalities of different classes of tasks are obvious, e.g., a master-slave application has two clusters), from clusters of previous phases in the same run (before a fault is manifested), or can be inferred from traces of previous normal runs. The erroneous tasks are those that deviate from the normal number of clusters, for example, by creating a separate cluster with few elements. Finally, AutomaDeD seeks to determine which code regions in the erroneous task(s) had the first manifestation of the error. For this purpose, AutomaDeD performs an activity called *edge isolation*, through which it identifies the SMM edge that contributed most to the distance of the erroneous task. In one mode of operation, multiple SMM edges, ranked by their contributions to the difference, are provided to the user. Developers can then target debugging effort to these code regions.

3. DESIGN OF AutomaDeD

In this section, we detail the design of the three novel aspects of AutomaDeD, which improves on the operational flow of the baseline that we have just described. In the improvements, we first show how edges in two graphs corresponding to two different tasks can be efficiently compared. Next, we show how the graphs corresponding to the SMMs can be compressed to improve the accuracy of identifying erroneous tasks and to reduce the computational cost of this process. Finally, we describe how identification of abnormal tasks is done in a scalable manner through two alternate methods of clustering and nearest-neighbor calculation.

3.1 Efficient Edge Comparison

In the error detection part, AutomaDeD performs pairwise comparisons of SMMs. In our previous work [8] we presented an algorithm to compute the dissimilarity (or difference) between a pair of SMMs. Its main idea was to find and add up the differences between corresponding edges in the two SMMs (that are being compared). An edge ($state_i$, $state_j$) that is present in two different SMMs, SMM_1 and SMM_2, implies that both tasks performed a state transition of the form $state_i \rightarrow state_j$ at some point in the program execution and we can compute edge differences. If an edge is present in one SMM but is not present in the other, the difference is assigned a high weight to highlight control flow differences that cause this behavior. Computing edge differences requires two steps to be computed efficiently:
(i) *Finding matching edges in two SMMs.* In order to compare an edge from an SMM, we must first determine whether a corresponding edge exists in the other SMM.
(ii) *Computing differences of edge attributes.* Once we have found corresponding edges, we compute the difference between their attributes, i.e., the differences between their transition probabilities and time probability distributions.

We efficiently perform the first step by representing SMMs using data structures that allow efficient edge searching. We use a sorted map of unique keys (a C++ map) to represent SMMs. The keys are edges and mapped data are edge attributes. This structure supports edge lookups with complexity O(log n), where n is the number of edges.

We represent SMM states by call stack *paths* collected when the program calls MPI routines. A call stack path is the list of function calls that are currently active, which includes the called functions and the offset into the functions. In previous versions of AutomaDeD, we used character strings to represent paths, which incurred a large overhead when we compared two states. Our current implementation supports a compact representation that uses unique references and, thus, supports direct comparison of the references. Further, since references from different tasks may point to the same path, we exchange the map between the two tasks to determine a consistent view of the paths before we compare their SMMs. This permits comparisons

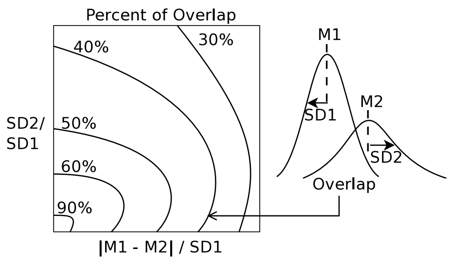


Figure 2: Table of the percent overlap of two normal distributions (assuming SD1 < SD2)

of edges through a few machine instructions (by reference comparisons) instead of by comparing character strings.

The second step is the edge-comparison computation. For this task we must calculate the difference between two time distributions—calculating the difference of the transition probabilities is trivial since it only involves a subtraction of two double-type values. We use the Lk-norm method [8] to compute the difference of two probability distributions, which is based on the following formula:

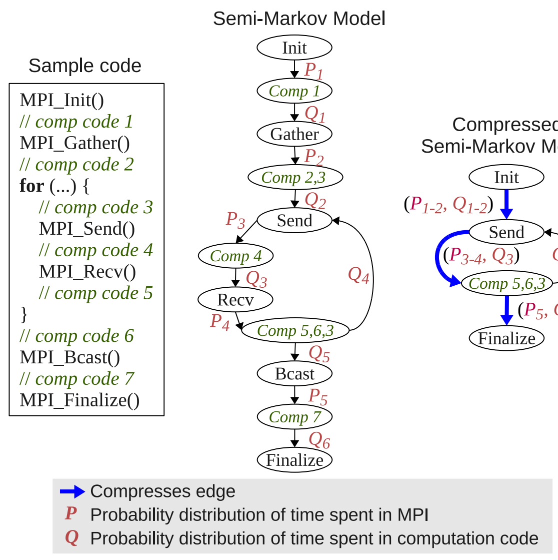
$$\int_{-\infty}^{\infty} |P(x) - Q(x)|^k dx, \qquad (1)$$

where $P(x)$ and $Q(x)$ are two continuous probability distributions of the random variable x (representing time) in the two edges. We estimate probability distributions with parametric and non-parametric methods. AutomaDeD uses normal distributions and histograms as the base models respectively. While histograms provide a better fit for the observed data than normal distributions, they have significantly higher memory and computational complexity. We therefore in the following use a normal distribution since we emphasize low overhead at scale.

Previous work in statistics [17] has shown that the overlap percentage of any two normal distributions can be estimated from their parameters, i.e., mean and standard deviation. A table (or nomogram) as shown in Figure 2 can be computed *a priori* so that we can obtain the overlap of two new distributions by inspecting the table, where the y-axis is the ratio of the largest standard deviation $SD2$ to the smallest standard deviation $SD1$, and the x-axis is the distance between the means $M1$ and $M2$ normalized by $SD1$. The key observation is that the Lk-norm of two normal distributions equals the area that does not overlap between the distributions. Therefore, AutomaDeD uses a similar table to estimate the value of an Lk-norm calculation without incurring in the high overhead of numerically estimating the integral in equation (1). The points in the table that AutomaDeD uses are calculated using the Lk-norm formula with parameters of two (randomly selected) normal distributions. Since we must quantize the parameters' values for which we store the results in the table, we only build a table of 500×500 values and approximate overlapping percents by interpolation. Our experiments show that a table of this size is sufficiently accurate to distinguish two normal distributions.

3.2 Graph Compression

Motivation. Parallel programs with complex control flow result in SMM graphs with many edges. For example, in our experiments with the NAS Parallel benchmarks, graph sizes



```
MPI_Init()
// comp code 1
MPI_Gather()
// comp code 2
for (...) {
    // comp code 3
    MPI_Send()
    // comp code 4
    MPI_Recv()
    // comp code 5
}
// comp code 6
MPI_Bcast()
// comp code 7
MPI_Finalize()
```

Figure 3: Compression approach.

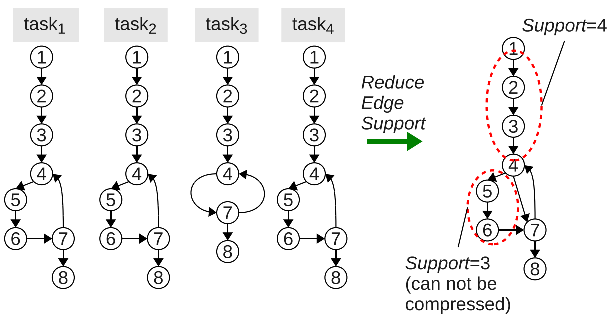


Figure 4: Global reduction of edges support.

are often on the order of hundreds of edges. Large edge counts impact the accuracy of isolation of abnormal tasks in AutomaDeD since the problem directly corresponds to the problem of detecting outliers (i.e., abnormal tasks) in a high dimensional space. High dimensional mathematical spaces create difficulties for unsupervised machine learning techniques such as clustering and k-nearest-neighbor due to the *curse of dimensionality* [7]. Distances between all pairs of points in high dimensional data tend to become almost equal—with too many dimensions, deviations from normality in a few dimensions are not as significant. Thus, AutomaDeD cannot find the abnormal task. An additional problem associated with SMM graphs of large sizes is that the overhead of the task isolation phase increases because the complexity of distance calculations of SMM pairs is proportional to the number of edges. Therefore, we implement an algorithm that allows AutomaDeD to compress large SMMs before we perform task isolation.

Which edges can be compressed. We observe that we can merge a linear chain of states and still retain the general control flow structure of the program, i.e., the states and edges that represent the main program loops are maintained in the compressed graph. Figure 3 illustrates this idea. The SMM represents the sample MPI code in the left part of the figure. We omit the transition probabilities associated with the edges (and only present the time distributions) for simplicity. The right part of the figure shows the compressed SMM after we apply our compression algorithm to the original SMM. We define a *sequence* of states as a linear chain of states with out-degree of one, in which the transition probabilities associated with their outgoing edges are 1.0. The compression algorithm merges sequences of states keeping the main control flow structure in the resulting compressed graph. The sequence of states after `Init` up until `Send` are summarized as only `Send`, and their edges are all merged into a single *compressed* edge.

A compressed edge contains two distributions P and Q that represent the time spent in the MPI calls and in the computation blocks of the original graph. Keeping time distributions separate in the compressed edge helps developers differentiate the parts of the code that are affected by MPI operations from parts in which the computation code in between them is the source of the problem.

How to assign attributes to compressed edges. We merge the time distributions in sequences of edges under the assumption that the underlying random variables are independent, and that normal distributions are used to fit the observed data. Given two independent random variables T_1 and T_2 that are normally distributed with parameters μ_1, σ_1 and μ_2, σ_2, the new random variable $T_1 + T_2$ is also normally distributed with parameters $(\mu_1 + \mu_2)$, $(\sigma_1 + \sigma_2)$. Therefore, when compressing a sequence of edges, the compression algorithm simply sums the parameters of the distributions in those edges, keeping distributions for communication (i.e., for MPI routines) and for computation blocks separate. As can be observed in Figure 3, a compressed edge has a tuple (P, Q), where P is the added distributions of communication states and Q is the added distributions of computation states.

Distributed nature of the merge process. The compression algorithm involves two steps; we now discuss our completely distributed implementation of both steps. First, AutomaDeD must determine the set of edges that are present in all the tasks or, if we have determined task equivalence classes, the set present within all tasks within an equivalence class. Our edge compression algorithm only targets such edges for compression. AutomaDeD does not compress edges that are not common in all SMMs to avoid eliminating abnormal transitions that may be present in only a few tasks. Second, the compression happens locally and concurrently at each task using the above set of edges.

Figure 4 illustrates this idea. We define an edge's *support* as the number of tasks in which it appears. We first apply a reduction operation that collects local information from all tasks and applies an aggregation operation on that information. The operation that the reduction performs sums the edge support of local graphs. At the end of the reduction, the edges of states 1–4 have a support of four (because they are present in all graphs), while edges of states 4–7 only have support of three. After we perform the reduction, the root process (i.e., the one that initiates the reduction) broadcasts the reduced graph to all tasks so that every task has the set of edges to compress, i.e. those with support of four for this example. We implement the reduction with a binomial tree, which has logarithmic complexity in terms of the number of tasks. We cannot use `MPI_Allreduce` since the tasks can contribute different numbers of edges.

```
DFSCompress(State state) {
  if (isNotFinalState(state)) {
    neighbors = getNeighbors(state)
    for each n in neighbors {
      Edge edge(state, n)
      if (edgeHasNotBeenVisited(edge)) {
        addEdgetoQueue(edge)
        if (isHeadOrTail(n)) {
          mergeEdgesInQueue()
          DFSCompress(n)
        }
      }
    }
  } else {
    mergeEdgesInQueue()
  }
}
```

Figure 5: Depth-first-search compression algorithm.

The second step compresses the local graph for edges that are globally supported. We use a modified version of the depth-first-search algorithm to traverse the graph. The algorithm's main idea is that sequences of states can be merged until the beginning or the end of a loop is found. The algorithm assumes an adjacency-list representation so it can find state neighbors for each state as the graph is traversed.

Figure 5 shows the compression algorithm's pseudocode. We define a *loop-head* state as the first state in a loop, and *loop-tail* state as the last state in a loop. As we traverse the graph, we store edges in a queue until the `mergeEdgesInQueue()` function is called in loop-heads, loop-tails or in the last state. This function merges edges as described previously in Figure 3 keeping probability distributions for communication and computation code regions separate. The complexity of the algorithm is O(*number of edges*).

Need for iterative drill-down due to graph compression. Due to graph compression, when AutomaDeD initially provides the characteristic edge(s) that likely caused the task to become anomalous, the granularity can be more coarse than in the baseline. The granularity can be a compressed edge, which includes multiple edges from the original graph. However, AutomaDeD keeps the original graph and the compressed graph in memory. For the task that we determine is anomalous, we perform edge isolation locally using the fragment of the original graph that corresponds to the part of the compressed graph that we determined is anomalous. For example, if edges $\{e_1, e_2, e_3\}$ were compressed into an edge $e^{(c)}$ and the initial iteration of the edge isolation flagged $e^{(c)}$ as the anomalous edge, the next iteration can work on $\{e_1, e_2, e_3\}$ and diagnose at the same granularity as baseline. We preserve the advantage of edge compression—AutomaDeD does not perform any communication of the potentially large original graph to other processes and does not have to perform task isolation on the original graph. The minor disadvantage is that edge isolation requires two iterations.

3.3 Scalable Outlier Detection

The typical use case of AutomaDeD isolates abnormal tasks in the time period the application fails. This can be challenging since AutomaDeD must extract a few abnormal tasks from many normal tasks. Naive techniques such as comparing each task against each other to find the most dissimilar task do not scale well since the complexity of these methods is quadratic with respect to the number of tasks.

We implement two scalable approaches to isolate abnormal tasks: (1) *clustering*, using CAPEK's algorithm, which first finds clusters and then determines abnormal tasks as indicated by the largest distances from their cluster centers; and (2) *nearest-neighbor*, which determines abnormal tasks based on the largest distances from their nearest neighbors. In both approaches, we sample a constant number of data points (i.e., tasks) and perform the analysis treating the sample set as representative of the entire set of points. Thus, we avoid having an unmanageable linear algorithmic complexity with respect to the number of points; both approaches scale with a complexity of the log of the number of tasks. Figure 6 illustrates the idea behind the two algorithms, which the next sections describe in detail.

3.3.1 Clustering

We have developed a novel outlier-detection technique based on CAPEK, a scalable clustering algorithm designed for large-scale, distributed data sets like those generated by parallel performance tools [10]. CAPEK finds groups, or *clusters* within distributed data sets, which gives us information about the structure of our data. For each cluster, CAPEK also determines a representative, or *medoid* m_i, and we can find outliers by finding the objects (SMM) in the data set that are furthest from their representative medoids.

Formally, CAPEK is a *K-Medoids* method. K-Medoids methods take a set of objects X, a dissimilarity function $d : X \times X \to \mathbb{R}$ and a number of clusters $k \in \mathbb{N}$ as input. They produce a *clustering*, a set of disjoint clusters $C = C_1, \ldots, C_k \subseteq X$ such that $\bigcup_{i=1}^{k} C_i = X$ and a set of medoids $M = m_1, \ldots, m_k$ such that $m_i \in C_i$. Each m_i is the representative element for cluster C_i. These methods attempt to minimize $\sum_{i=1}^{k} \sum_{x_j \in C_i} d(x_j, m_i)$, the total distance from each object to its representative medoid. The basic version of CAPEK requires the user to specify the number of clusters, k. It also allows the user to search for an ideal k using the Bayesian Information Criterion (BIC) [22]. The intuition behind this algorithm is that the medoids, m_i, will be approximately centered within their clusters, and they are thus good representatives for the clusters as a whole.

CAPEK has several advantages that make it well-suited for clustering SMM data. First, CAPEK is sampled, from which it derives its massive scalability. Traditional sequential clustering algorithms have quadratic or linear runtime, but CAPEK uses all processors for analysis to achieve logarithmic runtime, which makes our analysis feasible at scale. Other algorithms, such as the hierarchical clustering that baseline uses [8], do not readily support sampling, and thus do not scale to the system sizes that CAPEK supports.

Second, unlike *K-Means* methods [9, 18, 19], K-Medoids does not require that we can define algebraic operations, such as addition and scalar division, on the data. K-Means methods discover the synthetic means, or *centroids*, of their clusters using these operations, but we cannot directly calculate a "mean" SMM.

Finally, K-Medoids methods produce flat partitions of the data, which simplifies outlier detection. With hierarchical clustering, for example, we must choose the level in a clustering tree to describe clusters and outliers best. However, this complicated process does not scale. With a flat partition, we can detect outliers using standard deviation, which

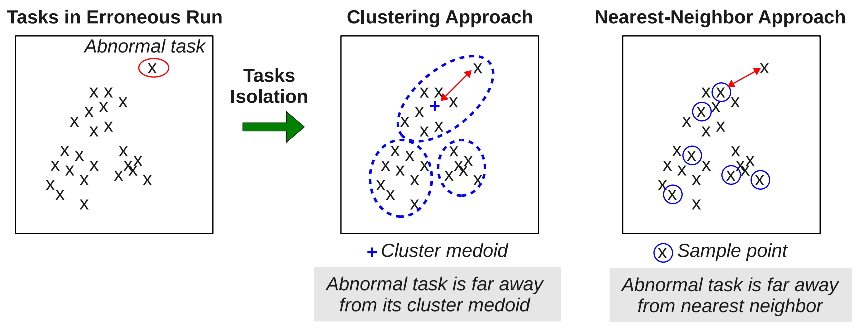

Tasks in Erroneous Run
Abnormal task
Tasks Isolation
Clustering Approach
+ Cluster medoid
Abnormal task is far away from its cluster medoid
Nearest-Neighbor Approach
Sample point
Abnormal task is far away from nearest neighbor

Figure 6: Clustering and Nearest-Neighbor methods to isolate abnormal tasks.

is straightforward and fast to compute. Our clustering-based outlier detection algorithm is:
(1) Perform clustering: We obtain a clustering C using CAPEK, which provides copies of the medoids m_i for all clusters to each process.
(2) Find distances from each task to its medoid: Each task computes the distance d_{ji} from its local SMM x_j to its representative, m_i. As CAPEK guarantees, this medoid will be the representative nearest to x_j.
(3) Normalize distances using standard deviation: Using parallel reductions within each cluster, we compute the standard deviation σ_i for each cluster in logarithmic time. We then normalize each process's d_{ji} to obtain $d'_{ji} = d_{ji}/\sigma_i$, which allows us to find outliers in data sets that may exhibit several different "normal" behaviors.
(4) Find top-k outliers: d'_{ji} is a measure of how far each SMM is from its representative; we now find the top k values of d'_{ji} in parallel. The corresponding SMMs are the "most different" from their representatives. We gather these SMMs, which we report as outliers. We can trivially find the top k SMMs in logarithmic time by computing k parallel reductions in sequence.

3.3.2 Nearest Neighbor

Our nearest-neighbor (NN) method classifies outlier tasks based on dissimilarities between tasks and their nearest neighbors. The main idea is that an abnormal task will be far from its nearest neighbor, while normal tasks will be close to each other so their pair-wise NN distances are small. To make NN scalable, we only perform NN distance calculations against a constant number of sample tasks, rather than against all tasks. When we calculate NN distances, a sample task removes itself from the sample tasks to avoid picking itself as its nearest neighbor. Our NN outlier detection algorithm is:
(1) Sampling: We locally generate a set of random indices that represent task ranks using a deterministic pseudorandom number generator with the same seed. After this step, each process can determine if it is a sample task.
(2) Broadcasting of samples: Each sample task broadcasts its SMM. After this step, each task can compare its SMM to the sample SMMs in order to find its NN distance.
(3) Find NN distance: Each task compare its SMM to those of the sample tasks. The SMM of the smallest distance corresponds to the NN task. Finally, we perform a global reduction at the root task to find the k most abnormal tasks.

NN may not isolate abnormal tasks if a fault affects multiple tasks simultaneously. To illustrate this problem, suppose that the normal clustering of the tasks in an application is two clusters, but a fault creates a third cluster with a few similar abnormal tasks. Suppose also that we sample two tasks t_1 and t_2 from this abnormal cluster. We will determine that t_1 and t_2 are each other's nearest neighbor. The distance values for each case will be low values and these tasks will not appear in the top-k rank of outliers (or abnormal tasks), which will prevent AutomaDeD from isolating them. The clustering approach avoids this problem because the abnormal tasks will belong to one of the normal clusters (because of the BIC methodology to select cluster configurations) and have high distances to the cluster medoid.

3.3.3 Abnormal Edge Isolation

Our previous work [8] presented mechanisms to detect the code region in which a fault is first manifested after task isolation. The *characteristic transition* is the edge that contributes most to the distance of the abnormal task. We easily extend this concept to multiple rank-ordered transitions ordered by their contributions to the difference. In this work we implement a modified version of one of these techniques. We perform the two outlier detection methods as follows:
(1) *Clustering*: After clustering, each task has the medoid of its cluster. We compare the abnormal task's SMM to the medoid's SMM and sort the edges in ascending order of their dissimilarities. We flag the top-k edges as abnormal.
(2) *NN*: After the task isolation phase, we compare the abnormal task SMM to all sample SMMs. As in the clustering method, we sort edges by their dissimilarities and flag the top-k edges as abnormal.

After we perform graph compression on the SMM (and have isolated the abnormal task), AutomaDeD keeps a copy of the original SMM in memory and performs edge isolation using that SMM. Since we must compare edges between graphs of the same nature—always between uncompressed graphs for this case—the abnormal task must have the original SMM of the other tasks. We fulfill this requirement by sending the original graph from the compared task(s) to the abnormal task. This additional step incurs a small overhead; for example, for the clustering method we only send one graph (from the medoid task) to the abnormal task.

4. EXPERIMENTS AND RESULTS

4.1 Fault Injection

We empirically evaluate the effectiveness of AutomaDeD's techniques by injecting faults that commonly occur in parallel applications. We inject faults into six applications of

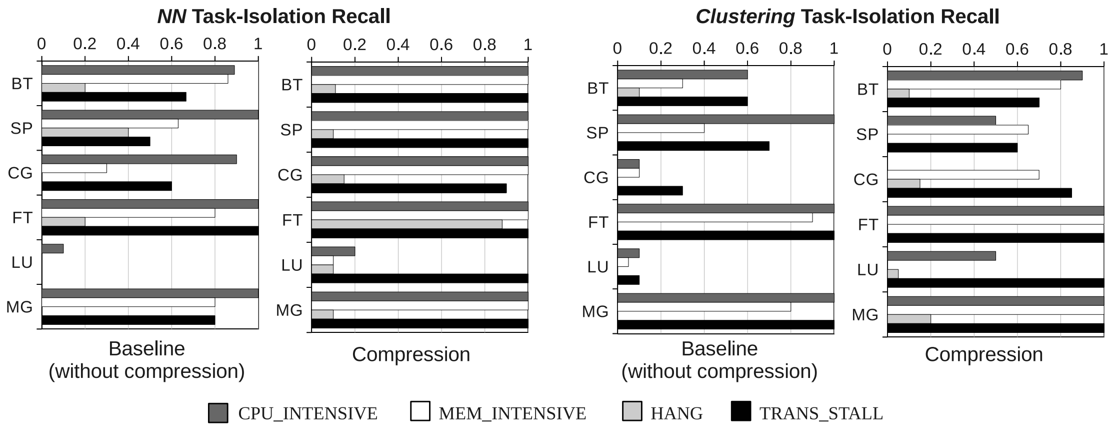


Figure 7: Task-isolation results for NN and clustering.

the NAS Parallel benchmark suite: BT, SP, CG, FT, LU and MG [6]. We omit EP because it performs almost no MPI communication and IS because it uses MPI only in a few code locations. Since their MPI profiles produce small SMMs, monitoring at the granularity of MPI calls does not suit these applications. Our injector [8] uses P^{N}MPI to inject a wide range of software faults into random MPI calls during MPI application runs. We focus our fault injection campaign on the following performance faults:

• `CPU_INTENSIVE`: CPU-intensive code region, emulated by a triply nested loop.
• `MEM_INTENSIVE`: Memory-intensive code region, emulated by filling a 1GB buffer with data at random locations.
• `HANG`: Local deadlock, emulated by making a process indefinitely suspend execution.
• `TRANS_STALL`: Transient stall, emulated by making a process suspend execution for 5 seconds.

In these experiments, we ran the benchmarks at a moderate scale: 512 processes for CG, FT, LU and MG, and 529 processes for BT and SP, with input size B. We use six-core nodes (the LLNL Sierra cluster), with 2.8 GHz Intel Xeon processors and 24 GB of RAM per node. Each experiment injects a single fault into a single random task during MPI communication operations (e.g., blocking and non-blocking sends and receives, all-to-all, broadcasts and barriers). For each benchmark, fault type, and detection technique, we perform 10 runs, for a total of 960 experiments.

4.2 Fault Injection Results

When we inject a fault, AutomaDeD performs the error-detection analysis by default at the end of the run during `MPI_Finalize`. For some benchmarks, the SMM created at the end of the run can be quite large, even after compression. For example, in LU and MG, the compression algorithm can only compress the graph to around 120 edges (from originally around 250 edges), which is still a large number of edges for the outlier detection techniques to work accurately. In these cases, AutomaDeD divides the run into phases to reduce the size of per-phase SMMs to a manageable size and performs the analysis in the faulty phase. In our prior work, we presented a technique to detect the abnormal phase in a run [8]. For these experiments, we assume that AutomaDeD is provided with the faulty phase, which the user can pinpoint or our prior algorithm can detect. In cases where a fault causes the application to suspend execution and it does not allow the creation of a new SMM at the end of a phase or during `MPI_Finalize`, such as in the `HANG` fault, the analysis is executed when AutomaDeD does not observe any state transitions for a "long" period of time; a parameter that can be configured by the user or estimated by AutomaDeD from previous runs (by taking the maximum transition time). In our experiments, we use 60 seconds for this parameter.

We use two metrics to evaluate error-detection and localization quality: *task-isolation recall*—the fraction of runs in which the task in which we inject the fault is in the top-5 abnormal processes that the task-isolation method (separately clustering or NN) outputs; *edge-isolation recall*—for cases in which AutomaDeD correctly isolates the abnormal processes, the fraction of runs in which the code region in which we inject the fault is in the top-5 abnormal edges.

Figure 7 shows the task isolation results for the baseline (without compression) and the compression approach, for the two outlier detection methods. Table 1 shows graph sizes for the baseline and the compression method for faults that do not suspend the execution of the program. When the program's execution is suspended due to a hang or segmentation fault, the size of the graph can vary depending on the number of transitions observed in the last snapshot time and is therefore not meaningful.

As we observe from Figure 7, compressing the SMM improves the accuracy of detecting the anomalous process in both the NN and the clustering methods. For example, in the NN method when we inject the `CPU_INTENSIVE` fault in BT, recall in the baseline approach is about 85% whereas with compression it is 100%. In the clustering method, for the same benchmark and fault, recall is improved from 60% in baseline to 90% with compression. Compression improves process-isolation recall because the dimensionality reduction that results from merging contiguous edges eliminates noisy (unimportant) dimensions from the outlier-detection analysis and allows AutomaDeD to focus its power on the significant dimensions.

These results also suggest that the NN method detects errors better than the clustering method for half of the tested benchmarks, while both perform equally well for the other half. However, we expect clustering to have better accuracy than NN for cases in which a fault manifestation affects more than one process, as previously discussed.

Benchmark	Original	Compressed	Comp. Ratio
BT	207	55	3.76
SP	179	55	3.25
CG	129	66	1.95
FT	21	4	5.25
LU	46	29	1.59
MG	81	33	2.45

Table 1: Edge counts for fault injection experiments.

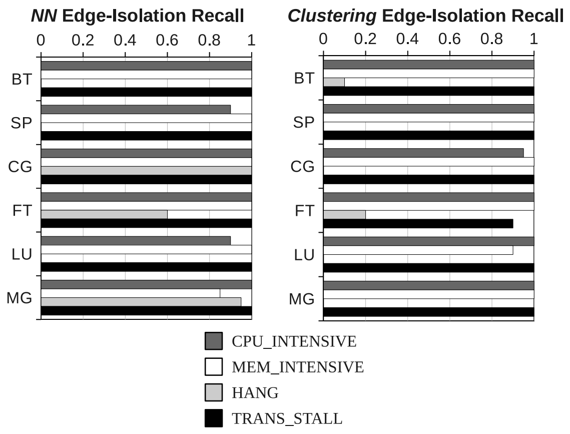


Figure 8: Edge-isolation with NN and clustering.

Figure 8 shows the edge isolation results. AutomaDeD provides a high overall edge-isolation recall for most injected faults. If AutomaDeD correctly detects the faulty task in the previous step, it can guide the developer, with a high accuracy, to the code region in which the fault first manifests itself as a timing abnormality. This positive result demonstrates that the compression of the graph does not affect this step, nor does the sampling approach that we used to make NN or clustering scalable. Thus, our sampling strategy is probably unbiased and provides representative samples.

4.3 Performance Results

We evaluate the performance improvement achieved by calculating Lk-norm values using a pre-computed table. Figure 9 shows times for the baseline case (estimating integral (1)) and the pre-computed case. We measure the time to calculate the dissimilarity between two SMMs while we vary the number of edges in the SMMs. The use of a pre-computed table improves performance significantly. For example, for two large SMMs of 1000 edges, the dissimilarity calculation takes 0.45 seconds when computing Lk-norms online, while it takes 10 milliseconds in the pre-computed case.

We evaluate AutomaDeD at larger scales and measure the time to perform the entire error-detection analysis ending in the edge isolation. We measure the individual times for each part of the analysis: compression, edge- and task-isolation, as we vary the number of tasks to more than 5,000. For this experiment we use the Algebraic MultiGrid (AMG) 2006 benchmark from the Sequoia benchmark suite [1], a scalable iterative solver and preconditioner for solving large structured sparse linear systems. We run experiments in the same cluster as our fault injection experiments. The analysis uses the SMM that corresponds to the entire execution of

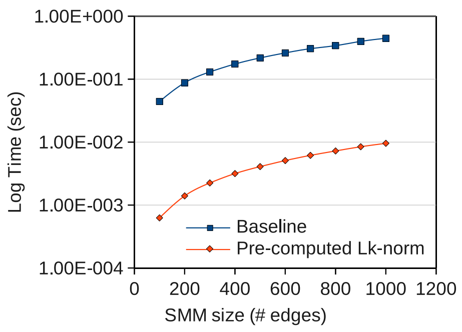


Figure 9: Lk-norm computation times.

AMG. Figure 10 shows the results of these experiments. The graph size without compression is (on average) 192 edges, and with compression is 158, with a compression ratio of 192/158 = 1.22.

As we observe from Figure 10, we can apply AutomaDeD at increasingly large scales with relatively little overhead for the error-detection analysis. For example, for runs of 5,832 tasks, the analysis takes less than 5 seconds for both the NN and the clustering methods. Graph compression only incurs a small overhead, on the order of 170 milliseconds for the largest runs. However it substantially improves the accuracy of error detection, as shown in the fault injection results (Figure 7). Compression requires little time because the core of the computation is performed locally in each process with a relatively small number of edges, (e.g., less than 250 edges for the AMG benchmark and the NAS Parallel Benchmarks). Despite compression decreasing edge support, this collective operation communicates little data since pairs of states in edges are represented as pointers (instead of strings as in our baseline).

The edge isolation step in NN (around 440 milliseconds) is larger than in the clustering method (around 4.5 milliseconds) because it compares edges of the abnormal processes to *multiple* sample SMMs, while the clustering method only compares edges of the abnormal process to one sample SMM, i.e., the medoid SMM. However, as the edge isolation results show, comparing edges to only one sample SMM suffices to produce a similar edge-isolation recall. These results demonstrate that the scalable techniques implemented in AutomaDeD make it suitable for online analysis in production runs. That is, AutomaDeD could be applied at multiple points in time as the application executes (possibly for several days) to find erroneous tasks and code regions.

To see the trend of the analysis time, we compute trend curves of the total time as Figure 11 shows. Logarithmic curves accurately model the observed data, which matches our expectation that the cost of our analysis scales logarithmically with system size. The algorithmic complexity of the techniques in the outlier detection step, which has an O(`log n`) scaling, dominate this cost. Evaluating the equations of the trend curves, the analysis would take 8.67 seconds for 10,000 tasks, and 11.29 seconds for 100,000 tasks, for the clustering method. While we realize that such extrapolations are problematic and not always accurate, they do show that there should be no inherent limits to scaling our approach and that AutomaDeD has the potential to be appropriate as an online tool at our target large scales.

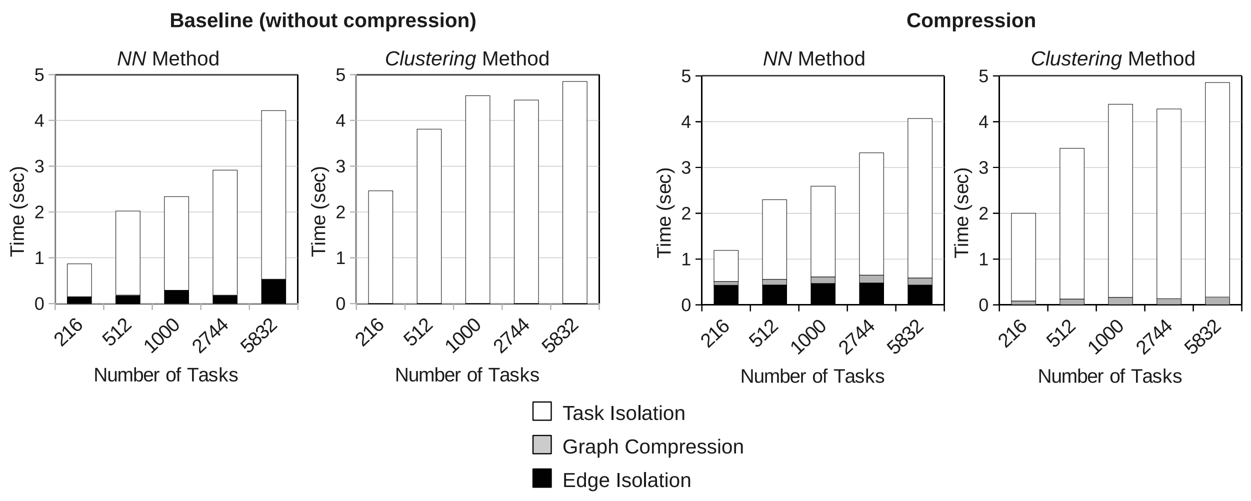


Figure 10: Time to isolate tasks and edges for the AMG2006 benchmark.

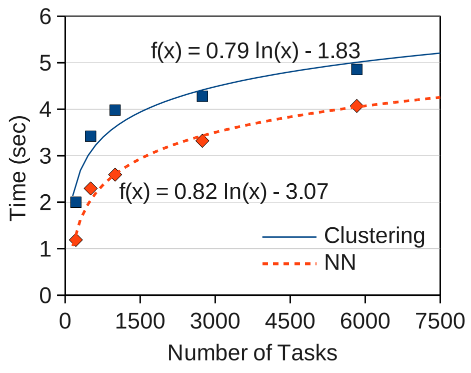


Figure 11: Trend lines for the total analysis time.

5. RELATED WORK

Traditional debugging techniques, including sequential debuggers such as gdb and "printf debugging," require that users manually trace the origins of their coding error. Traditional parallel debuggers, such as DDT [5] and TotalView [25], extend these techniques to allow tracing of multiple processes. They provide convenient interfaces to the state of these processes, but the main procedure of identifying errors remains manual. Overall, the traditional techniques require a significant amount of user experience, intuition and time, and thus are largely ineffective for debugging of large, complex parallel applications.

Several debugging tools detect bugs in large-scale applications without relying on much manual effort. These typically focus on detecting violations of deterministic and statistical properties of the applications. Deterministic tools can validate certain properties at runtime; any violation of these properties during an execution is reported as an anomaly. For example, FlowChecker [12] focuses on communication-related bugs in MPI libraries that these applications use. It extracts information on the application's intentions of message passing (e.g., by matching MPI Sends with MPI Receives) and at runtime checks whether the data movement conforms to these intentions. Bug localization follows directly: the data movement function that caused a discrepancy is the location of the bug.

Statistical tools [8, 11, 20] detect bugs by deriving the application's normal behavior and looking for deviations from it. For example, if the behavior of a process is similar to the aggregate behavior of a large number of other processes, then it is considered correct and different behaviors are considered incorrect. AutomaDeD [8] and Mirgorodskiy et al. [20] both monitor the application's timing behaviors and focus the developer on tasks and code regions that exhibit unusual behaviors. These tools focus on function call traces to identify the trace that is most different from other traces. AutomaDeD provides a more flexible framework for representing the application's timing behavior based on SMMs and does not assume that all processes behave identically, while Mirgorodskiy et al. can incorporate developer-provided information into their analysis. DMTracker [11] uses data movement related invariants, tracking the frequency of data movement and the chain of processes through which data moves. It builds invariants that describe normal data movements and signals alerts when it detects a data movement that does not conform to them.

While the above tools are effective in their own domains, their primary weakness is that their designs do not consider scalability. Typically, these tools collect trace data during the application's execution and write it to a central location. They then process the data to detect potential problems. In contrast, our system analyzes the application's behavior online, without any central bottlenecks. In this respect the closest prior work to ours is STAT [4, 15, 16], which provides scalable detection of task equivalence classes based on the functions that the processes execute. STAT uses MRNet [23], a tree-based overlay network, to gather and to merge stack traces across tasks and presents the traces in a call-graph prefix tree that identifies task equivalence classes. STAT removes problems associated with a central bottleneck by reducing the trace data as part of a computation being performed within the overlay network through a custom reduction plug-in. STAT focuses primarily on the state of the application once an error manifests itself whereas we focus on scalable analysis of the entire application execution.

6. CONCLUSION

We have implemented novel techniques in AutomaDeD that enables it to achieve scalability by optimizing it at different levels of its procedures. First, we minimize the time to compare elements of task models by using efficient data structures and approximation methods. Second, we reduce the sizes of the models to an appropriate magnitude, which eliminates noisy dimensions when finding the task affected by a fault. Finally, we use sampling-based techniques such as CAPEK's clustering and scalable nearest neighbor to deal with the increasing number of parallel tasks that are present in today's largest systems. Our implementation scales easily to thousands of tasks and it can identify erroneous tasks and code regions in a few seconds. With this performance, AutomaDeD can be used not only in debugging runs, but also in production runs in an online manner in which AutomaDeD's analysis would be applied periodically as the application runs (e.g., at boundaries of application phases) to detect problems automatically.

For future work, we will explore capturing more application information to detect a wider range of faults such as memory-related problems. We will also explore mechanisms to automate the detection of abnormal phases of execution in a scalable way by making use of previous correct runs of the same application.

7. REFERENCES

[1] Algebraic MultiGrid (AMG) 2006 Benchmark. `https://asc.llnl.gov/sequoia/benchmarks`.

[2] DynInst - An Application Program Interface (API) for Runtime Code Generation. `http://www.dyninst.org/`.

[3] The GNU C Library. `http://www.gnu.org/`.

[4] D. H. Ahn, B. R. D. Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable Temporal Order Analysis for Large Scale Debugging. In *Supercomputing Conference*, 2009.

[5] Allinea Software. Allinea DDT the Distributed Debugging Tool. `http://www.allinea.com/index.php?page=48`.

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

[7] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

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

[9] E. W. Forgy. Cluster Analysis of Multivariate Data: Efficiency *vs.* Interpretability of Classifications. *Biometrics*, 21:768–769, 1965.

[10] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Clustering Performance Data Efficiently at Massive Scales. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 243–252, New York, NY, USA, 2010. ACM.

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

[12] Q. Gao, W. Zhang, and F. Qin. FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking. In *ACM/IEEE Supercomputing Conference (SC)*, 2010.

[13] GDB Steering Committee. GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/documentation/`.

[14] V. E. Henson and U. M. Yang. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002.

[15] 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 *ACM/IEEE Supercomputing Conference (SC)*, pages 1–9. IEEE Press, 2008.

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

[17] J. M. Linacre. Overlapping Normal Distributions. *Rasch Measurement Transactions*, 10(1):487–8, 1996.

[18] S. P. Lloyd. Least Squares Quantization in PCM. Technical Note, Bell Laboratories. *IEEE Transactions on Information Theory*, 28:128–137, 1967, 1982.

[19] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. Univeristy of California Press, June 21-July 18 1967.

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

[21] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem Diagnosis in Large-Scale Computing Environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[22] D. Pelleg and A. Moore. X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734, 2000.

[23] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *SC '03*, 2003.

[24] M. Schulz and B. R. de Supinski. P^NMPI Tools: A Whole Lot Greater than the Sum of Their Parts. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

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