Debugging High-Performance Computing Applications at Massive Scales

Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski,
Todd Gamblin, Gregory L. Lee, Martin Schulz
Lawrence Livermore National Laboratory
{ilaguna, ahn1, bronis, tgamblin, lee218, schulzm}@llnl.gov

Saurabh Bagchi, Milind Kulkarni, Bowen Zhou
Purdue University
{sbagchi, milind, bzhou}@purdue.edu

Feng Qin
The Ohio State University
qin@cse.ohio-state.edu

1. INTRODUCTION

Breakthroughs in science and engineering are made increasingly today with the help of high-performance computing (HPC) applications. From understanding the process of protein folding to estimating short- and long-term climate patterns, large-scale parallel HPC simulations are the tool of choice. These applications can run detailed numerical simulations that model the real world. Given the great public importance of such scientific advances, the numerical correctness and software reliability of these applications is a major concern for scientists.

Debugging parallel programs is significantly more difficult than debugging serial programs—human cognitive abilities are overwhelmed when dealing with more than a few concurrent events [12]. When debugging a parallel program, a programmer must check the state of multiple parallel processes and reason about many different execution paths. The problem is exacerbated at large scale, when applications run on today's top supercomputers with millions of concurrently executing processes. Traditional debugging tools scale poorly with massive parallelism—they must orchestrate the execution of a large number of processes and collect data from them efficiently. The push toward exascale computing has heightened the need for scalable debugging techniques.

This article describes the fundamental advances made in designing scalable debugging techniques and shares the authors' experiences in applying their solutions to current petascale supercomputers. We present a set of techniques that build upon one another to accomplish large-scale debugging: discovery of scaling bugs (i.e., those that manifest themselves when the application is deployed at a large scale), behavioral debugging by modeling control-flow behavior of tasks, and software defect detection at the communication layer. We also describe support tools that are critical to higher-level debugging tools, and how these tools may evolve toward the exascale era. We discuss the three broad open problems in this domain: programmability challenges; performance bugs; and detecting silent data corruptions.

2. BACKGROUND

Source of bugs. Most large-scale HPC applications use the Message Passing Interface (MPI) to communicate among parallel processes in a distributed-memory model. The MPI standard provides more than 300 functions [18], including data transfer operations (such as send and receive for point-to-point transfer, and broadcasts for collective transfer), and process synchronization operations (such as barriers). Many functions have subtle semantics, such as pairing of calls in different processes. For example, a bug can be introduced by incorrectly copying a buffer from the application to the MPI library, or by sending a message on one process without properly calling a receive function on a peer.

Software bugs originate not only in the application, but also in the MPI library itself. Many MPI library implementations exploit obscure details of the standard to make aggressive optimizations to improve performance, and this can lead to numerical non-deterministic behavior. For example, a computation may produce different results each time it executes. Or, the MPI library might misuse a lower-level communication API (e.g., an interconnect API) and corrupt data while moving it between buffers in the application and the MPI library.

Scaling bugs. An insidious class of bugs specific to large-scale parallel programs only manifests when the application is run at large scale. We call these "scaling bugs". Scientific simulations are generally modeled and tested at small scale first. Then, when scientists believe the application is correct, they submit campaigns of large-scale production runs (e.g., with millions of parallel processes). Since resource constraints often limit testing and parallel program behavior is often scale and input dependent, scaling bugs are often not encountered before these campaigns. Most debugging tools in HPC, including traditional breakpoint-based debugging, and relative debugging tools struggle at such massive scales. Although advances in relative debugging allow scalable comparisons of program runs [6], the process of finding bugs remains fundamentally manual. Most HPC centers provide access to commercial debuggers like TotalView® and DDT®, which have been optimized for medium to large-scale parallel applications; however, further research is required to develop *more effective, scalable, and less manual* debugging methods like those presented in this article.

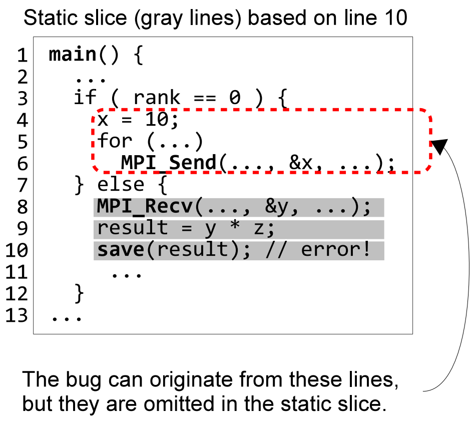
Scaling bugs appear for various reasons—overflow errors, resource exhaustion, and sub-optimal communication paths are the most common—and they can lead to correctness or performance errors. Figure 1 shows a sample bug, that appeared in the implementation of the `MPI_Allgather` routine in the MPICH [10] MPI implementation. Each parallel process gathers information from its peers using different communication topologies (e.g., ring, balanced binary tree) depending on the total amount of data to be exchanged. For a large enough scale an overflow occurs in the (internal) variable that stores the total size of data exchanged (see the line with the `if` statement). Consequently, a non-optimal communication topology is used. The only symptom the user of the simulation sees is a slowdown or *performance bug*. Determining that the root cause of this bug is a conditional overflow in a single MPI call is much more difficult.

Consequences of software defects. Software defects in HPC applications cause various levels of inconvenience. Hangs and crashes are common bug manifestations and their causes can be difficult to pinpoint because tight communication dependencies between processes are characteristic of HPC applications—an error in one process spreads quickly to others. Race conditions and deadlocks are challenging because they often become visible at large scales, where messages are more likely to interleave in different, untested orderings. Performance degradation leads to sub-optimal use of the expensive supercomputing facilities, which impacts the power budget at HPC facilities and the rate at which scientific questions can be answered. Numerical errors are among the most perplexing and deleterious, as they directly affect the validity of scientific discoveries. Scientists should be equipped with effective, easy-to-use debugging tools to isolate these problems quickly.

Complementary approach: Static analysis. Static analysis debugging tools can catch a variety of defects, but are insufficient to debug large-scale parallel computing applications. They lack runtime information, such as exchanged messages among processes, which is a major disadvantage as this information can be crucial in understanding the propagation of errors. To illustrate this problem, consider *program slicing*, a widely used static-analysis technique in debugging and software testing [23]. Program slicing uses data-flow and control-flow analysis to identify a set of program statements, the *slice*, that may affect the value of a variable at some point of interest in the program. Given an erroneous statement, developers use slicing to identify code that could have (potentially) caused that statement to fail. Consider Figure 2, in which an MPI program has an error in statement 10 when saving the results of some computation. The figure shows a static slice on lines 8–10, because the `result` variable, which is used in line 10, depends on the statements 9 and 10. However, due to the SPMD (single program multiple data) nature of MPI applications, one must take into account data propagation among processes, and highlight lines 3–6 as possible sources of the bug. This can only be done via dynamic analysis techniques—our main focus.

Complementary approach: Formal analysis. Formal analysis of MPI programs [9] can detect a variety of bugs. For example, ISP [21] provides formal coverage guarantees against deadlocks and local safety assertions.

A major limitation of such tools is that rule checking incurs high overheads due to state explosion. In addition, in current analysis implementations, MPI message scheduling



Static slice (gray lines) based on line 10

```
1  main() {
2    ...
3    if ( rank == 0 ) {
4      x = 10;
5      for (...)
6        MPI_Send(..., &x, ...);
7    } else {
8      MPI_Recv(..., &y, ...);
9      result = y * z;
10     save(result); // error!
11       ...
12   }
13 ...
```

The bug can originate from these lines, but they are omitted in the static slice.

Figure 2: Debugging example using static analysis.

is performed in a centralized engine, which limits scalability. The accuracy of such tools can be sacrificed to achieve better scalability for specific checks. However, these approaches have been shown to work only up to a few thousand processes, thus, more research is required to increase their capability for larger process counts. To detect a wider spectrum of bugs, formal analysis tools can be coupled with runtime MPI checkers such as Umpire [22], Marmot [13], and MUST [11]. However, the debugging process is still predominantly manual and they only cover bugs in the application, and not in the MPI library.

2.1 Approach Overview

In this article, we focus on dynamic techniques to detect bug manifestations (*i.e.*, software errors) in parallel applications and MPI. Our techniques aid the developer by pinpointing the root cause of the error at varying granularities.

Discovering scaling bugs. A challenge in developing large scale applications is finding bugs that are latent at small scales of testing, but manifest themselves when the application is deployed at a large scale. A scaling bug can manifest itself either on strong or weak scaling. Statistical techniques do not work well because they require comparison with error-free runs at the same scale as when the problem manifests itself, and such error free runs are often unavailable at the large scale. Later, in an expanded section, we describe recent developments to deal with scaling bugs.

Behavior-based debugging. These techniques rely on the observation that, although large scale runs involve a large number of processes, these processes can usually be grouped in only a few behavioral groups. We leverage this to develop techniques that reduce the search space—from thousands of processes to a few process groups—to help the developer in the bug hunting process.

The manifestation of a bug could be such that the *behavior* a few of the processes is different from the majority of the processes. Thus, abnormal behavior can be detected by identifying the abnormal processes. Later, we present `AutomaDeD`, a tool that automatically finds abnormal processes in MPI applications. `AutomaDeD` models control-flow and the timing behavior of each MPI process as a Markov model. These models are used to pinpoint suspicious processes and code regions in a scalable manner.

The experience of the authors in dealing with bugs at

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

    /* Use recursive doubling algorithm for small messages*/
  }
  ...
}
```

Figure 1: Sample scaling bug - in the MPICH2 library, caused by an overflow in the variable when run at large process counts or with a large amount of data.

large scale is that, when errors occur at large scale, only a limited number of behaviors is observed on processes, with only a few following the erroneous path where the fault is first manifested. Later, we present the Stack Trace Analysis Tool (STAT), which highlights these behaviors by attaching to all processes in a large-scale job, gathering stack traces, and merging the stack traces into a prefix tree to identify which processes are executing similar code.

Software defects in MPI. Although MPI is widely used in large-scale HPC applications and its implementations are high quality in general, MPI library implementations have suffered from software bugs, especially when ported to new machines. Many of these bugs are subtle and hard for the average programmer to detect [5]. Due to the wide use of MPI libraries, such bugs may have an impact on many parallel jobs. Later in the article, we present FlowChecker, a technique to detect communication-related bugs in MPI libraries. FlowChecker checks whether messages are correctly delivered from sources to destinations by comparing the flow of messages in the library against the program intentions.

In our experience, debugging at massive scales builds on many support diagnosis techniques. Developers must understand the scopes and strengths of these techniques and compose them to isolate the root cause quickly. Researchers are in the process of improving composability and workflow among these tools via smarter user interfaces and interoperable infrastructures.

3. DISCOVERING SCALING BUGS

As discussed earlier, scaling bugs are difficult to handle using traditional debugging techniques, and they are becoming more prominent with the incessant drive to execute HPC applications at massive scales. The goal is to detect such bugs early and to narrow down the originating code.

Zhou et al. developed an early attempt to retool traditional statistical debugging techniques to scaling bugs [24]. Their approach builds a statistical model from training runs that are expected *not* to be affected by the bug. The model captures the expected behavior of a program with simple control flow statistics (e.g., the number of times a particular branch is taken, or the number of times a particular calling context appears). Then, at deployment time, these statistics are measured at different points in the execution; if their values fall outside the model parameters, the deployed run is deemed to manifest a bug since it is be different from the training runs. Examination of the program features that deviate then localizes anomalies to find potential bugs.

This approach failed to find scaling bugs due to a simple but fundamental reason. If the statistical model is trained only on small-scale runs, statistical techniques can result in numerous false positives. Program behavior naturally changes as the level of concurrency increases (*e.g.*, the number of times a branch in a loop is taken will depend on the number of loop iterations, which can depend on the scale). Small scale models will incorrectly label correct behaviors at large scales as anomalous (and therefore erroneous). This effect can be particularly insidious with strong scaling, where scaling up divides work among more and more processes, and each process does progressively less work.

The most successful techniques for handling scaling bugs built in scale as a parameter of the model and relied on the availability of bug-free runs at small scales [24, 25]. **Vrisha**, a framework that uses many small-scale training runs, is an example of such techniques. **Vrisha** builds a statistical model (based on Kernel Canonical Correlation Analysis, or KCCA) from these runs to infer the relationship between scale and program behavior. In essence, the technique builds a scaling model for the program that can extrapolate the aggregated behavioral trend as the input or system size scales up. Bugs can be detected automatically by identifying deviations from the trend. Notably, this detection can occur *even if the program is run at a previously unmeasured scale.* The schematic of this approach is shown in Figure 3.

However, Vrisha can only identify that the scaling trend has been violated; it cannot determine which program feature violated the trend, nor where in the program the bug manifested. The WuKong framework solved this by making the observation that if all features are combined and modeled (as Vrisha did), these cannot be "reverse-mapped" to identify which feature caused the deviation. Therefore, **WuKong** uses per-feature regression models, built across multiple training scales that can accurately predict the expected bug-free behavior at large scales. When presented with a large-scale execution, **WuKong** uses these models to infer what the value of each feature *would have been in a bug-free*

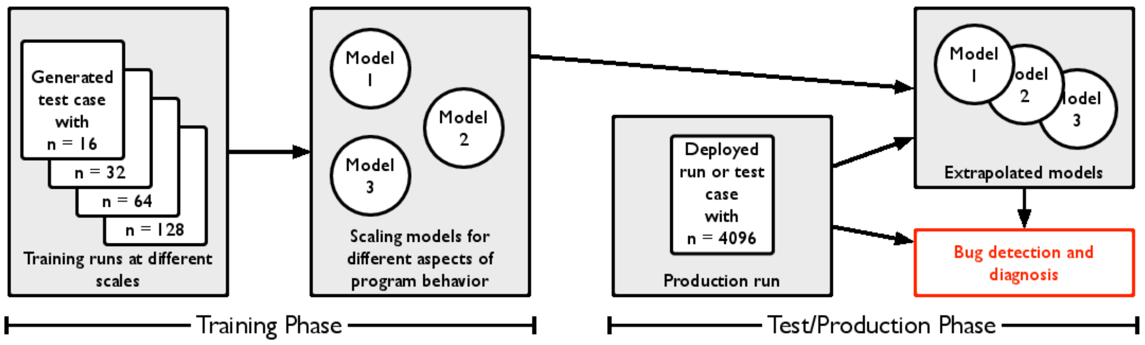

Generated test case with n = 16
n = 32
n = 64
n = 128
Training runs at different scales
Model 1
Model 2
Model 3
Scaling models for different aspects of program behavior
Deployed run or test case with n = 4096
Production run
Model 1
Model 2
Model 3
Extrapolated models
Bug detection and diagnosis
Training Phase
Test/Production Phase

Figure 3: An overview of the approach to detect and diagnose scaling bugs. It relies on using scale as a model parameter and training using error-free runs at small execution scales.

run. If any value deviates from the prediction, `WuKong` detects a bug. `WuKong` identifies the lines of code that result in unexpected behavior by ranking features based on their prediction error—program features are carefully chosen so that they can be linked to particular regions of code. This ranked list provides a roadmap the programmer can use to track down the bug.

Scaling models cannot predict program behaviors (or features) that do not correlate with scale. These techniques use cross-validation techniques to prune features that are hard to model accurately from the training runs. This approach limits the behaviors that they can predict.

4. BEHAVIOR-BASED DEBUGGING

An application bug may manifest in such a way that the behavior of a few buggy parallel tasks is different from that of the majority of the tasks. We leverage this observation to focus the developer's attention on the few buggy tasks, rather than requiring an exhaustive investigation. Our approach builds a *behavioral model* for tasks that allows us to compare the behavior of one task with another. To isolate bugs on the largest supercomputers effectively, we need scalable distributed algorithms that can isolate anomalous behaviors quickly. We have implemented all of these in our `AutomaDeD`[1] framework [4, 15].

`AutomaDeD` uses a simple model of task behavior that captures timing information and patterns in each task's control flow. The timing information allows us to detect performance problems, and the control flow model allows us to isolate them to particular code regions. Figure 6 shows an overview of `AutomaDeD`. On each concurrent task, a measurement tool builds a model of execution at the granularity of code blocks and execution paths between them. To make our model lightweight, we do not model basic blocks, but code regions *between* communication (MPI) calls. The tool uses intercepts MPI calls dynamically, and builds its model of each task in these instrumentation functions.

Each task's behavior is stored as a semi-Markov model (SMM). States in the model represent communication code regions (i.e., code inside MPI routines), and computation code regions (i.e., code executed between two MPI communication routines). A state consists of a call stack trace, sampled at each MPI call entry. The stack trace gives the dynamic calling context of each communication call. SMM

[1]**Automa**ta-based **De**bugging for **D**issimilar parallel tasks

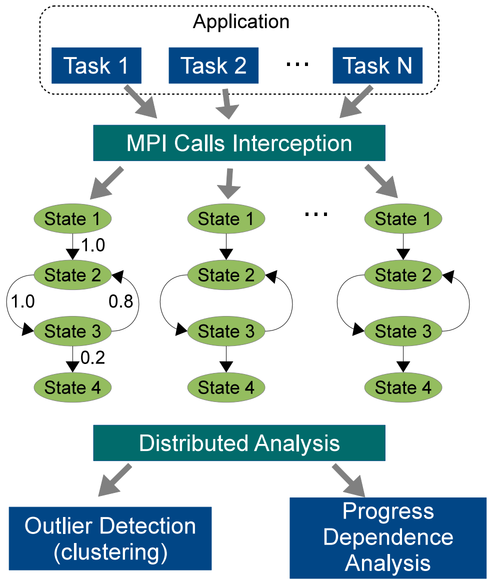


Figure 4: Overview of the `AutomaDeD` framework.

edges represent transfer of control between two states. The modeling strategy assigns two attributes to each edge: (1) a transition probability, and (2) a distribution of execution times. The distribution models the time spent in the source state for each dynamic invocation where control next transferred to the destination state.

Given per-task models, `AutomaDeD` identifies anomalous behaviors using a clustering algorithm, which takes as input a set of per-task models and a (dis)similarity metric that can compare two models. The algorithm outputs groups of tasks with similar models. Clusters of tasks are considered unusual if they are not expected based on the clusters in error-free training runs, or error-free iterations of the same run (in an unsupervised setting). For example, consider a master-slave parallel application, which, in an error-free run, can be clustered naturally into two behavioral clusters: a master and a slave cluster. If `AutomaDeD` finds a third cluster in a faulty run, `AutomaDeD` flags it as unusual and focuses attention on the the tasks in the cluster. A similar algorithm identifies the state (or code region) in which the error first manifests in those tasks. We have found that other tech-

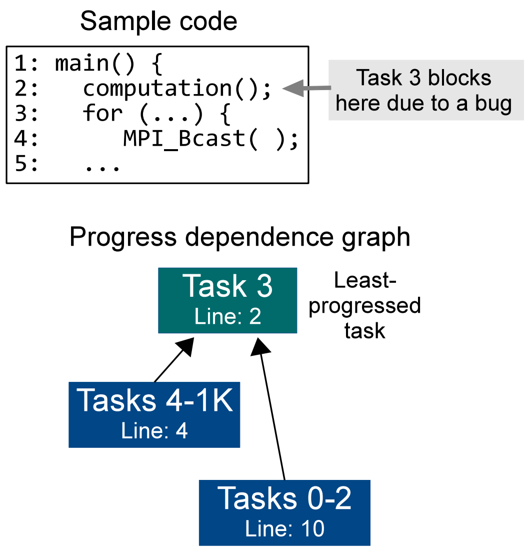


Figure 5: Progress-dependence graph.

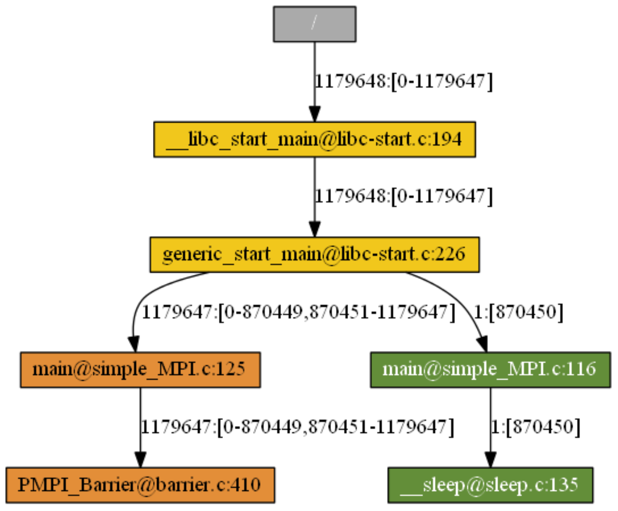


Figure 6: View of STAT for a run on more than 1 million MPI processes.

niques to identify the task outliers are also useful, such as finding tasks that are abnormally far from its cluster center, and identifying tasks that are on average far from their neighbors using nearest-neighbor algorithms [15].

Hangs and deadlocks are common bug manifestation in HPC applications, and particularly difficult to diagnose at massive scale. Large-scale HPC applications are tightly coupled (e.g., a MPI collective operation involves the participation of multiple tasks) thus, a fault in one task quickly propagates to other tasks. We have developed the concept of a *progress dependence graph* (PDG) of tasks that captures the partial ordering of tasks with respect to their progress toward the final computation result [14, 19]. The PDG is then used to determine the *least-progressed* task, which is the likely root cause of the problem. Markov models in `AutomaDeD` are used to create the PDG. Dependencies on progress are calculated based on how the MPI tasks visit different application states. For example, suppose that two tasks x and y are blocked on different states, S_x and S_y respectively. If y has visited S_x before (and no path from y to x exists in the model), y probably depends on x to make progress and an edge is created in the PDG from y to x. Figure 5 a snippet of a PDG in which there the are three classes of tasks (0-2; 3; 4-1,000). Task 3 has made the least progress according to the PDG and will then be examined further for the root cause of the fault.

The combination of local, per-task models and distributed outlier and progress analysis allows `AutomaDeD` [15] to focus developers' attention on the erroneous period of time, parallel task, and specific code regions that are affected by faults and performance problems in parallel applications. This focus substantially reduces the bug search space and relieves the user of the burden of comparing potentially millions of tasks and even more possible execution paths.

All detection and diagnosis analysis steps in `AutomaDeD` (both outlier detection and progress dependence) are performed in a distributed fashion, and thus scale to large numbers of parallel tasks [15]. Outlier task isolation uses scalable clustering [7] and progress dependence is calculated using a combination of binomial tree-based reductions and per-task local analysis. The complexity of these procedures with respect to the number of parallel tasks is, in the worst case, logarithmic. `AutomaDeD` has been used to diagnose the origin of real-world bugs that appeared only at large scales. In a previous article [14], we have detailed how we used `AutomaDeD` to identify the offending task, when a hang manifested in a molecular dynamics application with more than 8,000 tasks on the BlueGene/L supercomputer.

5. STACK TRACE ANALYSIS TOOL

The Stack Trace Analysis Tool (`STAT`) [3] is a lightweight and highly scalable debugging tool for identifying errors in code running on the world's largest supercomputers. `STAT` gathers stack traces from a parallel application and merges them into a call-graph prefix tree. `STAT`'s merge operation groups together traces from different processes that have the same calling sequence and labels the edges of these merged traces with the count and set of MPI rank processes that exhibited that calling sequence. Further, nodes in the prefix tree that are visited by the same set of processes are given the same color, providing the user with a quick means of identifying the various process equivalence classes.

In fact, process equivalence classes are `STAT`'s main debugging idiom to help users focus only on a manageable number of processes. Bulk-synchronous applications, particularly in a hang state, typically demonstrate a small number of differently behaving classes, with a few processes taking an anomalous call path, a similarly small number of processes waiting for point-to-point communication, and the remaining processes stuck in a collective operation. Thus, users can effectively debug a large-scale application, even with scales over one million MPI processes [16], by focusing on a small subset of the tasks, in particular, a single representative of each equivalence class.

While conceptually simple, `STAT` is effective in isolating highly elusive errors that often emerge in production computing environments. In addition, the development, deployment and use of this tool provided us with a blueprint for scalable tools. On extreme-scale systems, we learned that an effective tool must scale in several dimensions. For one,

A Real-World Debugging Case

A scientist experienced hangs in a laser-plasma interaction code (named PF3D) when he scaled it to 524,288 MPI processes on LLNL's Sequoia BlueGene/Q system. He spent months trying to debug the problem using print statements to no avail, and he was reluctant to run a traditional parallel debugger at such a large scale. Furthermore, he was unable to reproduce the hang at smaller scales, where these fully-featured, heavyweight debuggers would be more plausible.

The scientist was made aware of STAT, and within a few minutes of launching the tool, he was able to identify that the application was deadlocked in a computation-steering module. The scientist and his colleagues easily analyzed STAT's output to determine that the hang was the result of a race condition that occurred between two distinct, but overlapping, communication regions. This problem occurred because PF3D was migrating from one version to another more scalable, but incompatible one. During this migration, PF3D was running through a compatibility layer, which introduced the race condition and ultimately caused these timing- and scale-dependent hangs. After this analysis, the user expedited the task of porting to the new version, after which the simulations were able to proceed without hangs.

STAT informed the user of the precise location of the hang in a few minutes, and a few hours of analysis determined the cause of the hang. By contrast, the user spent months of effort, and perhaps millions of CPU hours, and was still unable to identify the problem using the common procedure of print debugging.

This case is just one example of many problems that STAT was able to debug at large scales, with other cases even extending beyond 1.5 million MPI processes and with problems ranging from user application errors to system software bugs and even hardware faults.

it must be capable of presenting large amounts of debug information without overwhelming the users. Equally importantly, it must become a highly scalable application on its own, collecting, managing and reducing debug data without suffering a scalability bottleneck.

To become a highly scalable application, from its inception `STAT` built on scalable infrastructures such as LaunchMON [2] for tool start-up and boostrapping, and the MRNet [20] tree-based overlay network for communication. Even then, as we have tested and deployed it on increasingly larger systems, scalability bottlenecks have continually surfaced. Thus, we have had continuous innovation to improve its key attributes, from its internal data representations [16], to file access patterns and our testing methods [17].

6. SOFTWARE DEFECTS IN MPI

Bugs in MPI libraries can be devastating in a production supercomputing environment, because the majority of parallel supercomputing applications depend on the MPI library. MPI bugs can especially frustrate application developers since they may appear to be bugs in client code. Further, these bugs can be tedious for library developers to catch and fix [1]. MPI users' machines may have different architectures or compilers from library developers' machines. Worse, the library developers may not have a system as large as the one on which the bug manifests. Certain bugs only occur on large-scale systems [3]. Other bugs may only manifest in large MPI applications, and if the applications are sensitive or proprietary, users usually have to generate a small reproducing test program to send to developers, which could be a time-consuming process. In many cases, library developers may simply be unable to reproduce a scaling bug.

`FlowChecker` is a low-overhead method for detecting bugs in MPI libraries [5] Its main idea is to check whether the underlying MPI libraries correctly deliver messages from the sources to the destinations as specified by the MPI applications. Like a delivery service's package tracking system, `FlowChecker`'s detection process includes extraction of message-passing intention (source and destination addresses), message flow tracking (package transmission and delivery), and message delivery verification (user confirmation).

Specifically, `FlowChecker` first extracts the intentions of message passing (*MP-intentions*) from the MPI applications. Normally, MP-intentions are implied by the MPI function calls and the corresponding arguments made in the MPI applications. For example, `FlowChecker` can extract MP-intentions based on a matched pair of `MPI_Send` and `MPI_Recv` function calls, which are collected by instrumenting the MPI applications.

Second, for each MP-intention, `FlowChecker` tracks the corresponding message flows by following the relevant data movement operations starting from the sending buffers at the source process. Data movement operations move data from one memory location to another within one process or between two processes [5, 8]. Examples of data movement include memory copy and network send/receive, which are collected by instrumenting the MPI libraries. This step allows `FlowChecker` to understand how the MPI libraries perform message transmission and delivery.

Finally, `FlowChecker` checks message flows (from the second step) against the MP-intentions (from the first step). If a mismatch is found, `FlowChecker` reports a bug and provides further diagnostic information such as faulty MPI functions or incorrect data movements.

Figure 7 illustrates the main idea of `FlowChecker`. In this example, process 1 invokes `MPI_Send` to send a message stored at the buffer {`A1`, `A2`} to process 2, while process 2 invokes `MPI_Recv` to receive the message and store the message at the buffer {`D1`, `D2`}. Here, the MP-intention is {`A1`→`D1`, `A2`→`D2`} and the message flows are `A1`→`B1`→`C1`→`D1` and `A2`→`B2`→`C2`→`D2'`. After comparing the MP-intention with the corresponding message flow, `FlowChecker` will detect that the data in `A2` is delivered to an incorrect destination `D2'`, instead of `D2` as specified by the application. Additionally, `FlowChecker` will provide further disgnosis information: the last broken message flow {`C2`→`D2'`}.

7. CONCLUSION

In this article, we argued that a robust set of dynamic debugging tools are required for deploying large-scale parallel applications. This work complements ongoing work in static analysis, relative debugging, and formal methods for development of correct parallel applications. We have surveyed the state-of-the-art techniques for dynamic debugging. The first class that we discussed targets software bugs that arise only when the application runs at large scales. The technique used is to extrapolate from behavior at small-scale runs that are assumed to be correct. The second class is behavior-based detection through clustering, where behav-

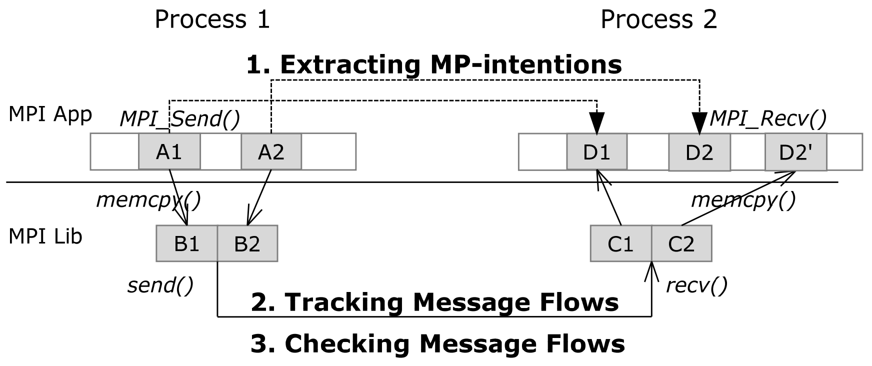


Figure 7: An example to illustrate the main idea of `FlowChecker`, which is used for finding bugs in the MPI communication libraries.

ior includes control flow and timing of a certain granularity of code regions. A sub-class within this collects stack traces from each individual process and then, using the insight that the stack traces exhibit great commonality, merges multiple traces into one, thus providing a developer with a limited and manageable amount of information. The third class targets bugs in the communication libraries, by verifying that the program intent matches the observed behavior. These software packages build on top of robust, usable tools, which include stack tracing, dynamic binary instrumentation, scalable data analytics (such as, clustering), and runtime profiling.

Looking ahead, discoveries are needed in three main research directions. *First*, higher level software development environments must be created for parallel programming, to enable faster development with fewer defects. Such environments should support widely recognized software engineering principles, such as, refactoring, code completion and quick fix support, and intuitive user interfaces. *Second*, debugging support must be deployed for performance bugs, in addition to the current focus of correctness bugs. For this purpose, application models must include performance measures and must support extrapolation of performance characteristics in various dimensions, such as, larger process counts, larger data sizes, and different data sets. *Third*, attention must focus on data corruptions that result from software bugs. These corruptions are often not within the purview of existing detection techniques, *i.e.*, they do not cause hangs or crashes or performance slowdowns, but "silently" corrupt the data output. These bugs lead to the specter to incorrect science, which surely we all wish to avoid.

The area of dynamic debugging techniques has spurred significant innovation and robust tool development. We will build on this promising base to solve further challenges, some of which we have laid out above.

Acknowledgments

This work was performed partially under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-JRNL-652400).

8. REFERENCES

[1] Open MPI Bug Tickets. https://svn.open-mpi.org/trac/ompi/ticket/689.

[2] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *Proceedings of the Internation Conference on Parallel Processing*, pages 578–585, 2008.

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

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

[5] Z. Chen, Q. Gao, W. Zhang, and F. Qin. FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking. In *Proceedings of the 2010 ACM/IEEE conference on Supercomputing (SC '10)*, Nov 2010.

[6] M. N. Dinh, D. Abramson, and C. Jin. Scalable relative debugging. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):740–749, March 2014.

[7] 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)*, pages 243–252. ACM, 2010.

[8] Q. Gao, F. Qin, and D. K. Panda. DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC '07)*, Nov 2007.

[9] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky. Formal Analysis of MPI-Based Parallel Programs. *Communications of the ACM*, 54(12):82–91, 2011.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[11] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for High*

Performance Computing 2009, pages 53–66. Springer, 2010.

[12] D. E. Kieras, D. E. Meyer, J. A. Ballas, and E. J. Lauber. Modern Computational Perspectives on Executive Mental Processes and Cognitive Control: Where to from Here. *Control of Cognitive Processes: Attention and Performance*, pages 681–712, 2000.

[13] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI Analysis and Checking Tool. *Advances in Parallel Computing*, 13:493–500, 2004.

[14] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin. Probabilistic Diagnosis of Performance Faults in Large-Scale Parallel Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 213–222. ACM, 2012.

[15] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Ahn, M. Schulz, and B. Rountree. Large Scale Debugging of Parallel Tasks with AutomaDeD. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, page 50. ACM, 2011.

[16] 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 conference on Supercomputing (SC '08)*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[17] 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 *Parallel Computing: Architectures, Algorithms and Applications (ParCo 2007)*, Julich/Aachen, Germany, 2007.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. `http://www.mpi-forum.org/docs/`, Sep 2012.

[19] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin. Accurate Application Progress Analysis for Large-Scale Parallel Debugging. In *ACM International Symposium on Programming Language Design and Implementation (PLDI)*, pages 1–10. ACM, 2014.

[20] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 21. ACM, 2003.

[21] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[22] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *ACM/IEEE 2000 Conference Supercomputing (SC '00)*, pages 51–51. IEEE, 2000.

[23] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[24] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: Using Scaling Properties of Parallel Programs for Bug Detection and Localization. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC)*, pages 85–96. ACM, 2011.

[25] B. Zhou, J. Too, M. Kulkarni, and S. Bagchi. WuKong: Automatically Detecting and Localizing Bugs that Manifest at Large System Scales. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 131–142. ACM, 2013.