Diagnosis of Performance Faults in Large Scale MPI Applications via Probabilistic Progress-Dependence Inference

Ignacio Laguna, *Member, IEEE*, Dong H. Ahn, *Member, IEEE Computer Society*, Bronis R. de Supinski, *Member, IEEE Computer Society*, Saurabh Bagchi, *Senior Member, IEEE*, and Todd Gamblin, *Member, IEEE*

Abstract—Debugging large-scale parallel applications is challenging. Most existing techniques provide little information about failure root causes. Further, most debuggers significantly slow down program execution, and run sluggishly with massively parallel applications. This paper presents a novel technique that scalably infers the tasks in a parallel program on which a failure occurred, as well as the code in which it originated. Our technique combines scalable runtime analysis with static analysis to determine the least-progressed task(s) and to identify the code lines at which the failure arose. We present a novel algorithm that infers probabilistically progress dependence among MPI tasks using a globally constructed Markov model that represents tasks' control-flow behavior. In comparison to previous work, our algorithm infers more precisely the least-progressed task. We combine this technique with static backward slicing analysis, further isolating the code responsible for the current state. A blind study demonstrates that our technique isolates the root cause of a concurrency bug in a molecular dynamics simulation, which only manifests itself at 7,996 tasks or more. We extensively evaluate fault coverage of our technique via fault injections in 10 HPC benchmarks and show that our analysis takes less than a few seconds on thousands of parallel tasks.

Index Terms—Distributed debugging, MPI, progress dependence, parallel applications

1 Introduction

Debugging errors and abnormal conditions in large-scale parallel applications is difficult. While high performance computing (HPC) applications have grown in complexity and scale, debugging tools have not kept up. Most debugging tools do not run efficiently on the largest systems. More importantly, they provide little insight into the causes of failures. Traditional debugging techniques allow programmers to inspect the state of parallel tasks over time, but the process of identifying the root cause of problems often requires substantial manual effort.

The question of identifying root cause is particularly challenging for performance faults (e.g., slow code regions) and correctness problems (e.g., deadlocks), which may manifest in a different code region or on a different process from their original cause. To provide insight into these problems, previous work [1], [2] identify the *least-progressed (LP)* task (or tasks) in a parallel execution. The LP task often corresponds to the faulty task, which helps programmers to narrow down their debugging efforts from inspecting the state of multiple tasks—possibly million of them—to one (or a few) task(s).

In contrast to [1], which identifies the LP task using static and dynamic analysis, the *AutomaDeD* [1] tool identifies it via low-cost dynamic analysis [2]. *AutomaDeD* probabilistically identifies the LP task (or tasks) using a Markov model (MM) as a summary of each task's control-flow history. States in the MM represent MPI calls and regions of computation in application code, and edges represent state transitions. The models are created online with little overhead and provide rich information that allows tracing performance and correctness problems to their origin.

AutomaDeD introduced the concept of *progress dependence* to probabilistically pinpoint the LP task given a faulty execution of the application [2]. It creates a *progress dependence graph* (PDG) to capture wait chains of non-faulty tasks that depend on the faulty task to progress. We use these chains to find the LP task. Once we find the LP task, *AutomaDeD* applies source-code analysis on this task's state to identify code that may have caused it to fail.

The original algorithm to create a PDG computes progress dependencies in a fully distributed manner by using per-task Markov models [2]. This algorithm incurs imprecision when calculating dependencies since only a partial view of the MM is seen by each task. MMs may be different among tasks (i.e., they may have different states and edges). For example, a task may reach a state in which an MPI message is sent, whereas another tasks may reach a (different) state, in which that message is received. Dissimilarities

- *I. Laguna, D. H. Ahn, B. R. de Supinski, and T. Gamblin are with the Lawrence Livermore National Laboratory, Livermore, CA 94550. E-mail: {ilaguna, ahn1, bronis, tgamblin}@llnl.gov.*
- *S. Bagchi is with the Department of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN 47907. E-mail: sbagchi@purdue.edu.*

Manuscript received 19 Sept. 2013; revised 12 Feb. 2014; accepted 19 Mar. 2014. Date of publication 20 Apr. 2014; date of current version 8 Apr. 2015. Recommended for acceptance by S. Aluru.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2314100

1. *Automa*ta-based *De*bugging for *D*issimilar Parallel Tasks.

among MM could cause impression when computing progress dependencies since dependencies could conflict with each other (e.g., in one task the dependence is in one direction, whereas in another task, the dependence is in the opposite direction).

In this paper, we present a novel algorithm to compute the PDG—and subsequently to find the LP task(s)—that uses a global MM as input, instead of local MMs. We compare this semi-distributed algorithm (named *SEMI-DIST*) with the original fully distributed algorithm (named *DIST*). *SEMI-DIST* is more precise than *DIST* for some applications at the expense of extra overhead. On the other hand, since *DIST* is fully distributed, it ensures scalability—it can be used to find the LP task in a fraction of a second among thousands of MPI tasks. Both algorithms use minimal per-task information and incur only slight runtime overhead. Our implementation is transparent—it uses the MPI profiling interface to intercept communication calls.

This paper presents a fault coverage evaluation of *DIST* and *SEMI-DIST* on ten HPC benchmarks. We show, through fault injection, that *AutomaDeD* constructs a PDG and finds a faulty task in a fraction of a second on each program running with up to 32,768 tasks. *AutomaDeD* accurately identifies the LP task in 95 percent of the cases (using *DIST*), averaged across the tested benchmarks. Its precision (i.e., the inverse of false-positive rate) is 90 percent on average using *SEMI-DIST*. In a blind study, we also show that *AutomaDeD* can diagnose a difficult-to-catch bug in a molecular dynamics code [3] that manifested only at large scale, with 7,996 or more tasks. *AutomaDeD* quickly found the origin of the fault—a complex deadlock condition.

This paper makes the following contributions:

- A novel scalable semi-distributed algorithm to create a progress-dependence graph and the LP task(s) to quickly diagnose the origin of performance faults.
- A comparison of accuracy and precision of the new algorithm (*SEMI-DIST*) with respect to the original one (*DIST*) in identifying the LP task(s) under fault injections in 10 HPC benchmarks.
- A study of scalability *DIST* and *SEMI-DIST* with up to 32,768 MPI tasks.

This paper extends our previous conference paper [2] by creating a semi-distributed algorithm that is more accurate but less scalable than the fully distributed version. We also present a more comprehensive evaluation across a variety of benchmarks.

The remainder of this paper is organized as follows: Section 2 presents the overview of our approach and Sections 3 and 4 detail our design and implementation. Section 5 presents our case study and fault injection experiments. In Sections 6 and 7, we survey related work and state our conclusions.

2 Overview of the Approach

2.1 Progress Dependence Graph

A *progress-dependence graph* represents dependencies that prevent tasks from making execution progress. A *dependence* is any relationship among two or more tasks that prevents the execution of one of the tasks from advancing. For example, a task might block while waiting for a message from another task. We use these relationships to find the causes of failures, such as program stalls, deadlocks, and slow code regions.

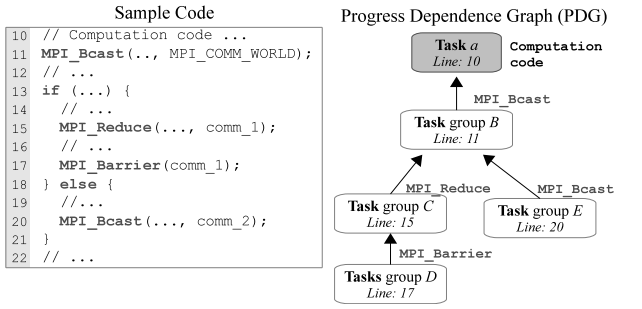

Fig. 1. Progress dependence graph example. Task a blocks in computation code which leads to groups of tasks B, C, D and E to block in other code regions, i.e., collective MPI calls. Task groups cannot progress due to task a.

A PDG starts with the observation that all tasks may need to enter an MPI collective call before some tasks can exit the call. For example, MPI_Reduce is often implemented in MPI using a binomial tree [4]. Since the MPI standard does not require collectives to be synchronized, some tasks could enter and leave this state—the `MPI_Reduce` function call—while others remain in it. Tasks that only send messages in the binomial tree enter and leave this state, while tasks that receive and later send messages block in this state until the corresponding sender arrives. These blocked tasks are *progress-dependent* on the other delayed tasks.

Definition 1 (Progress dependence). *Let the set of tasks that participate in a collective operation be X. If a task subset $Y \subseteq X$ has reached the collective operation while another task subset $Z \subseteq X$, where $X = Y \cup Z$, has not yet reached it at time t, such that the tasks in Y blocked at t waiting for tasks in Z, then Y is progress-dependent on Z, which we denote as $Y \overset{pd}{\rightarrow} Z$.*

Fig. 1 shows a sample PDG, in which task a blocks in computation code on line 10. Task a could block for many reasons, such as a deadlock due to incorrect thread-level synchronization. As a consequence, a group of tasks B block in `MPI_Bcast` in line 11 while other tasks proceed to other code regions. Task groups C, D and E block in code lines 15, 17, and 20, respectively. No progress dependence exists between groups C and E, and between groups D and E, because they are in different branches of execution.

2.1.1 Point-to-Point Operations

In blocking point-to-point operations such as `MPI_Send` and `MPI_Recv`, the dependence is only on the peer task.

Definition 2 (Point-to-point progress dependence). *If task x blocks when* sending *(receiving) a message to (from) task y at time t, then $x \overset{pd}{\rightarrow} y$.*

This definition also applies to nonblocking operations such as `MPI_Isend` and `MPI_Irecv`. The main difference is that the dependence does not apply directly to the send (or receive) operation, but to the associated completion (e.g., a wait-loop or test operation). If a task x blocks on `MPI_Wait`, for example, we infer the task y, on which x is

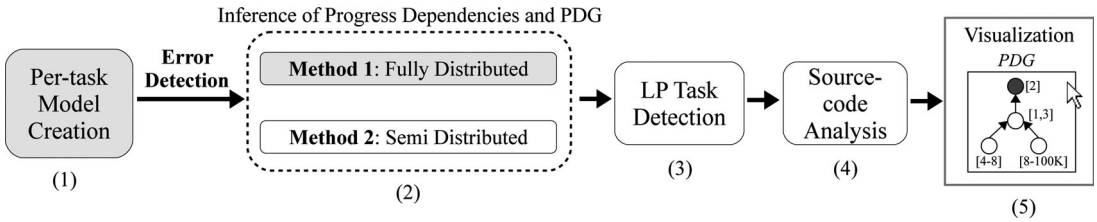

Fig. 2. Diagnosis work flow (gray blocks execute in a distributed fashion).

progress-dependent, from the request on which x waits. Similarly, if x spins on a test (e.g., by calling `MPI_Test` within a loop), we can infer the peer task on which x is progress-dependent from the associated request. On the receiving end, we can also infer the dependence from other test operations such as `MPI_Probe` or `MPI_Iprobe`. In any case, we denote the progress dependence as $x \xrightarrow{pd} y$.

2.1.2 PDG-Based Diagnosis

A PDG can intuitively pinpoint the task (or task group) that initiates a performance failure. In Fig. 1, task a can be blamed for causing the stall since it has no progress dependence on any other task (or group of tasks) in the PDG. It is also the least-progressed task. From the programmer's perspective, the PDG provides useful information for debugging and performance tuning. First, given a performance failure such as that in Fig. 1, the PDG shows where to focus attention, i.e., the LP task(s). Second, we can efficiently apply static or dynamic bug-detection methods based on the state of LP task(s). *AutomaDeD* applies slicing [5] starting from the state of the LP task, which substantially reduces the search space when compared to traditional slicing methods.

2.2 Workflow of Our Approach

Fig. 2 shows the workflow of the approach.

1. *Markov-model creation*. *AutomaDeD* captures per-MPI-task control-flow behavior in a Markov model. MM states correspond to two code region types: *communication* regions (i.e., code executed within an MPI function), and *computation* regions (i.e., code executed between two MPI functions).
2. *Inferring dependencies and PDG creation*. When a performance fault is detected, *AutomaDeD* creates a PDG. We provide two algorithms to create a PDG. The first algorithm is distributed (which we have referred to earlier as *DIST*): it computes a local PDG in each task (using task local information) and then performs a global reduction to create the final PDG. The second algorithm is semi-distributed (which we have referred to earlier as *SEMI-DIST*): it first performs a reduction to compute a global MM in a single task—a distributed operation—and then computes the final PDG based on this MM in a single task. It has better precision than the the first algorithm, and almost the same accuracy, at the cost of extra overhead.
3. *LP-task detection*. Based on the reduced PDG, we determine the LP task and its state, which we use in the next step (i.e., source-code analysis). The LP task(s) is(are) simply a group of tasks without any progress dependence.
4. *Source-code analysis*. The state of the LP task allows us to apply a variety of static or dynamic source-code analysis techniques to backtrack to the fault's origin. We perform (backward) program slicing since it allows *AutomaDeD* to identify code that could have led the LP task to reach its current (faulty) state.
5. *Visualization*. Finally, *AutomaDeD* presents the program slice, the reduced PDG and its associated information.

3 Design

In this section, we present the inference of progress dependence based on Markov models.

3.1 Summarizing Execution History

AutomaDeD captures a compressed version of control-flow behavior in each task using an MM. States represent communication and computation states, whereas edge weights capture the frequency of transitions between two states. Fig. 3 illustrates how *AutomaDeD* creates MMs at runtime. We use the MPI profiling interface to intercept MPI routines. Further details on the MM creation process can be found in the online supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2314100.

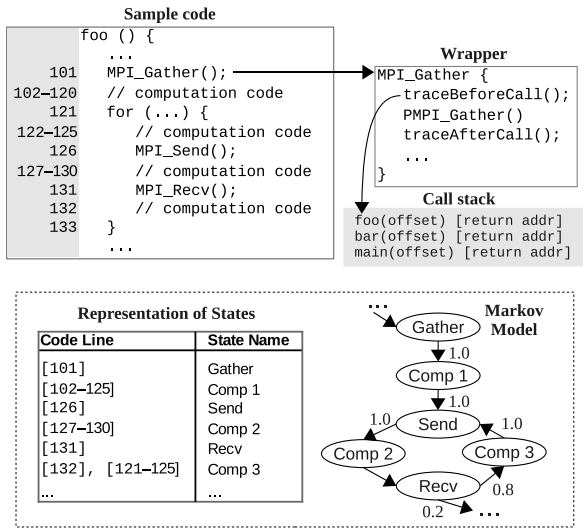

Fig. 3. Markov model creation. States represent code regions within or between MPI calls. Edge annotations represents the frequency of transitions between states.

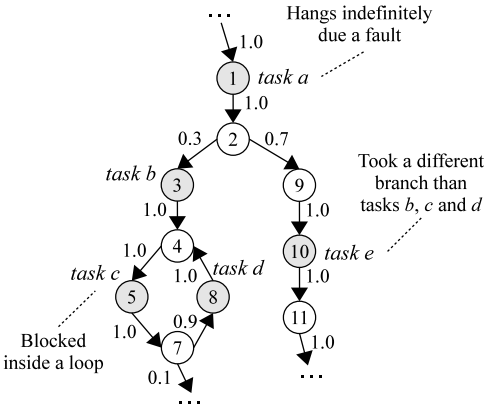

Fig. 4. Sample Markov model with five tasks that blocked in different states.

3.2 Progress Dependence Inference

We now discuss how we infer progress dependencies probabilistically from our MMs. We restrict the discussion to dependencies that arise from collective operations, since dependencies from point-to-point operations do not require our probabilistic analysis. For example, if task t_i is waiting for another task in `MPI_Recv`, *AutomaDeD* uses the parameters of the MPI call to determine the task on which t_i's progress depends. When a task blocks in `MPI_Wait`, for example, when using non-blocking operations, *AutomaDeD* uses MPI request handles to identify the matching progress-dependent task. We cannot infer progress dependence locally in a task t_i that blocks in an `MPI_ANY_SOURCE` request—*AutomaDeD* simply omits a progress-dependence edge in this case. However, our progress-dependence analysis can still find dependencies *from* other tasks to t_i (e.g., from tasks waiting on the subsequent global barrier), thus, the state of t_i is not eliminated from the PDG. In cases where t_i is the LP task, we can often pinpoint it accurately since typically the rest of the tasks block on later MPI calls, such as collectives, which usually create dependencies from those states to t_i.

AutomaDeD probabilistically infers progress dependence between a task's local state and the states of other tasks. Intuitively, our MM models the probability of going from state x to state y via *some* path $x \rightsquigarrow y$. If a task t_x in x must eventually reach y with high probability then we can determine that a task t_y in state y could be waiting for t_x, in which case we infer that $y \xrightarrow{pd} x$. For simplicity, we represent progress dependencies in terms of task states (rather than in terms of task IDs) in the rest of the discussion—we assume that a task t_x can only be in one state at the same time, i.e., state x.

To illustrate how progress dependence is calculated, we introduce the concept of path probability:

Definition 3 (Path probability). *The path probability of two states is the sum of the probabilities of moving from one state, the source, to another state, the destination, via all possible paths in a given MM. A path probability, p, can be either forward or backward, and* $0.0 \leq p \leq 1.0$.

Definition 4 (Forward and backward path probability). *A forward path probability between states x and y is the path probability where x is the source and y is the destination, i.e., $P(x, y)$. A backward path probability between states x and y is the path probability where y is the source and x is the destination, i.e., $P(y, x)$.*

TABLE 1
Dependence Based on Path Probabilities

$P(i,j)$			$P(j,i)$				
0	$0 < P < 1$	1	0	$0 < P < 1$	1	Dependence?	Type
✓			✓			No	
✓				✓		Yes	$t_i \xrightarrow{pd} t_j$
✓					✓	Yes	$t_i \xrightarrow{pd} t_j$
	✓		✓			Yes	$t_i \xleftarrow{pd} t_j$
	✓			✓		?	
	✓				✓	Yes	$t_i \xrightarrow{pd} t_j$
		✓	✓			Yes	$t_i \xleftarrow{pd} t_j$
		✓		✓		Yes	$t_i \xleftarrow{pd} t_j$
		✓			✓	?	

Undefined dependencies are denoted as "?."

Fig. 4 illustrates how we infer progress dependence from the MMs. Note that this figure shows a global view of the MM and the current states of all the tasks. However, locally each task only has a partial view of this graph. For example, task e, which is in state 10, only sees the right branch of the graph, which begins from state 2 and continues to state 9.

In Fig. 4, five tasks (a, b, c, d, and e) are blocked in different states (1, 3, 5, 8, and 10, respectively). To estimate the progress dependence between tasks b and c, we calculate the path probability $P(3, 5)$, the probability of going from state 3 to state 5 over all possible paths, which is 1.0. Thus, *task c is likely to be waiting for task b, since the observed execution dictates that a task that is in state 3 must always reach state 5*. To estimate progress dependence more accurately, we consider the possibility of loops and evaluate the backward path probability $P(5, 3)$, which in this case is zero. Thus, task c cannot reach task b, so we can consider it to have progressed further than task b. We can now infer that $c \xrightarrow{pd} b$.

3.2.1 Resolving Conflicting Probability Values

When a forward path probability $P(i, j)$ is 1.0 and a backward path probability $P(j, i)$ is zero, a task in state j has made more progress than a task in state i. However, if the forward path probability $P(i, j)$ is 1.0 and the backward path probability is nonzero then the task in state j might return to i. For example, for tasks d and c in Fig. 4, $P(8, 5) = 1.0$ but $P(5, 8) = 0.9$. In this case, task d must eventually reach state 5 to exit the loop, so we estimate that $c \xrightarrow{pd} d$; our results demonstrate that this heuristic works well in practice most of the time. The dependence between task b and task e is null: no progress dependence exists between them. The same is true for the dependencies between task e and task c or d.

3.2.2 General Progress Dependence Estimation

To estimate the progress dependence between tasks t_i and t_j in states i and j, we calculate two path probabilities: (i) a forward path probability $P(i, j)$; and (ii) a backward path probability $P(j, i)$. We use Table 1 to estimate progress dependencies. If both probabilities are zero (i.e., the tasks are in different execution branches), no dependence exists between the tasks. When one probability is 1.0 and the other

is less than 1.0, the first *predominates* the second. Therefore, the second probability determines the dependence. For example, if the second is $P(j,i)=0$, then $t_j \overset{pd}{\rightarrow} t_i$ since executiongoes from i to j. Similarly, if one probability is zero and the second is nonzero, then the second predominates the first. Therefore, the first probability determines the dependence. For example, if the first is $P(i,j)=0$, then $t_i \overset{pd}{\rightarrow} t_j$ because execution could go from j to i but not from i to j.

We cannot determine progress dependence for two cases: (1) when both probabilities are 1.0, and (2) when both probabilities are in the range $0.0 < p < 1.0$. The first case could happen when two tasks are inside a loop and, due to an error, they do not leave the loop and block inside it. In this case both backward and forward path probabilities are 1.0, so it is an undefined situation. The probabilities in the second case simply do not provide enough information to make a decision. For these cases, *AutomaDeD* marks the edges in the PDG as undefined so the user knows that the relationship could not be determined. These cases occurred infrequently in our experimental evaluation. When they do, the user can usually determine the LP task by looking at tasks that are in one group or cluster. Section 5 gives examples of how the user can resolve these cases visually.

AutomaDeD constructs a local PDG in each task. The procedure to construct this PDG can be found in the online supplemental material.

4 Scalable Mechanisms

This section details our PDG analysis implementation that ensures scalability with increasing numbers of MPI tasks. In particular, we give more details of the *DIST* and *SEMI-DIST* algorithms to build the PDG.

Before triggering the PDG-based analysis, we assume that a performance problem has been detected. *AutomaDeD* provides a timeout mechanism for detection. Details about this mechanism can be found in the online supplemental material.

4.1 Inference of the PDG

The following procedures assume that progress dependencies are found from MMs. If MMs do not have any state, we cannot construct a PDG. However, an MM is empty only if no MPI calls have been made—the first MM state is `MPI_Init`—which could occur due to job initialization problems. However, the focus of *AutomaDeD* is on performance problems, which usually occur after the initialization phase.

4.1.1 Distributed Algorithm

Algorithm 1 provides more detail of Step 2 in our workflow. We first perform a reduction over the current state of all tasks to compute the $statesSet$ of all tasks. We next broadcast $statesSet$ to all tasks. Each task uses algorithm 3 to compute its local version of the PDG from its local state and $statesSet$. Finally, we use a parallel reduction of the local PDGs to calculate the union of the edges (forward or backward dependencies).

When reducing a PDG, the associative operation applied is the union of progress dependencies. Table 2 shows examples of some union results given two progress dependencies from two different tasks. In case 1, a dependency is present in only one task so the dependency predominates. In cases 2 and 3, the dependencies are similar so we retain it. In case 4, they conflict so the resulting dependency is undefined. We efficiently implement this operator using bitwise `OR` since we represent dependencies as integers.

TABLE 2
Examples of Dependence Unions

No	Task x	Task y	Union	Reasoning	OR operation
1	$i \rightarrow j$	null	$i \rightarrow j$	first dependence dominates	1 + 0 = 1
2	$i \rightarrow j$	$i \rightarrow j$	$i \rightarrow j$	same dependence	1 + 1 = 1
3	$i \leftarrow j$	$i \leftarrow j$	$i \leftarrow j$	same dependence	2 + 2 = 2
4	$i \rightarrow j$	$i \leftarrow j$	$i?j$	undefined	1 + 2 = 3
5	null	null	null	no dependence	0 + 0 = 0

Algorithm 1 Distributed PDG computation

Input: $currState$: current state of the task
Output: $matrix$: adjacency-matrix representation of PDG

```
1: procedure DIST
2:     statesSet ← Reduce currState to rank 0
3:     statesSet ← Broadcast statesSet from rank 0
4:     matrix ← call LocalPDG to build PDG
5:     matrix ← Reduce matrix to rank 0
6: end procedure
```

4.1.2 Semi-Distributed Algorithm

A shortcoming of the previous algorithm (*DIST*) is that progress dependencies are found based on a single task's view of the Markov model—different tasks could have different MMs so this could lead to inaccuracies when building the PDG (e.g., case 4 in Table 2). We present a second algorithm in which a global MM is created first and then the PDG is computed based on it. The algorithm is named *SEMI-DIST* since it is partially distributed—the local MM is still created in a distributed manner.

Algorithm 2 Semi-distributed PDG computation

Input: mm: Markov model
$currState$: current state of the task
Output: $matrix$: adjacency-matrix representation of PDG

```
1: procedure SEMI-DIST
2:     globalMM ← Reduce mm to rank 0
3:     statesSet ← Reduce currState to rank 0
4:     if rank is 0 then
5:         matrix ← call LocalPDG to build PDG
6:     end if
7: end procedure
```

The complexity analysis of both algorithms as well as diagrams that illustrate their main workflow can be found in the online supplemental material.

4.2 Determination of LP Task

We compute the LP task from the reduced PDG. *AutomaDeD* first finds nodes with no outgoing edges based on dependencies from collectives and marked them as LP. If more than one node is found, *AutomaDeD* discards nodes that have point-to-point dependencies on other non-LP tasks in different branches. Since *AutomaDeD* operates on a probabilistic framework (rather than on deterministic

methods [1]), it can incorrectly pinpoint the LP task (e.g., when forward and backward probabilities are both zero), although such errors are rare according to our evaluation. However, in most of these cases, the user can still determine the LP task by visually examining the PDG (by looking for nodes with only one task).

4.3 Guided Application of Program Slicing

4.3.1 Background

Program slicing transforms a large program into a smaller one that contains only statements that are relevant to a particular variable or statement. For debugging, we only care about statements that could have led to the failure. However, message-passing programs complicate program slicing since we must reflect dependencies related to message operations.

We can compute a program slice statically or dynamically. We can use static data and control flow analysis to compute a static slice [5], which is valid for all possible executions. Dynamic slicing [6] only considers a particular execution so it produces smaller and more accurate slices for debugging.

Most slicing techniques that have been proposed for debugging message-passing programs are based on dynamic slicing [7], [8], [9]. However, dynamically slicing a message-passing program usually does not scale well. Most proposed techniques have complexity at least $O(p)$. Further, the dynamic approach incurs high costs by tracing each task (typically by code instrumentation) and by aggregating traces centrally to construct the slice. Some approaches reduce the size of dynamic slices by using a global predicate rather than a variable [8], [9]. However, the violation of the global predicate may not provide sufficient information to diagnose failures in complex MPI programs.

We can use static slicing if we allow some inaccuracy. However, we cannot naively apply data-flow analysis (which slicing uses) in message-passing programs [10]. For example, consider this code fragment:

```
1  program() {
2    ...
3    if (rank == 0) {
4      x = 10;
5      MPI_Send(...,&x,...);
6    } else {
7      MPI_Recv(...,&y,...);
8      result = y * z;
9      printf(result);
10     ...
```

Applying traditional slicing on the `result` statement in line 9 identifies statements 7, 8, and 9 as the only statements in the slice, however, statements 3-9 should be in the slice. The slice should have statements 4-5 because the value `x` sent is received as `y`, which influences `result`. Thus, we must consider the SPMD nature of the program in order to capture communication dependencies. The major problem with this *communication-aware slicing* is the high cost of analyzing a large dependence graph [10] to construct a slice based on a particular statement or variable. Further, MPI developers must decide on which tasks to apply communication-aware static slicing since applying it to all tasks is infeasible at large scales.

4.3.2 Using Slicing in AutomaDeD

AutomaDeD progressively applies slicing to the execution context of tasks that are representative of behavioral groups, starting with the groups that are most relevant to the failure based on the PDG. *AutomaDeD* uses the following procedure:

1. Initialize an empty slice S.
2. Iterate over PDG nodes from the node corresponding to the LP task to nodes that depend on it, and so on to the leaf nodes (i.e., the most progressed tasks).
3. In each iteration i, $S = S \bigcup s_i$ where s_i is the statement set produced from the state of a task in node i.

5 Evaluation

We demonstrate how *AutomaDeD* diagnoses a difficult bug in a molecular dynamics program and evaluate *AutomaDeD* extensively in a controlled setting by performing fault injection in ten MPI benchmarks.

5.1 Case Study

An application scientist challenged us to locate an elusive error in ddcMD, a parallel molecular dynamics code [3]. The bug manifested as a hang that emerged intermittently only when run on Blue Gene/L with 7,996 MPI tasks. Although the developer had already identified and fixed the error with significant time and effort, he hoped that we could provide a technique that would not require tens of hours. In this section, we present a blind case study, in which we were supplied no details of the error, that demonstrates *AutomaDeD* can efficiently locate the origin of faults.

Fig. 5 shows the result of our analysis. Our tool first detects the hang condition when the code stops making progress, which triggers the PDG analysis to identify MPI task 3,136 as the LP task—*AutomaDeD* first detects tasks 3,136 and 6,840 as LP tasks and then eliminates 6,840 since it is point-to-point dependent on task 0, a non-LP task, in the left branch. The LP task in the `a` state, causes tasks in the `b` state that immediately depend on its progress to block, ultimately leading to a global stall through the chain of progress dependencies. This analysis step reveals that task 3,136 stops progressing as it waits on an `MPI_Recv` within the `Pclose_forWrite` function. Once it identifies the LP task, *AutomaDeD* applies backward slicing starting from the `a` state, which identifies `dataWritten` as the data variable that most immediately pertains to the current point of execution. Slicing then highlights all statements that could directly or indirectly have affected its state.

The application scientist verified that our analysis precisely identified the location of the fault. ddcMD implements a user-level, buffered file I/O layer called `pio`. MPI tasks call various `pio` functions to move their output to local per-task buffers and later call `Pclose_forWrite` to flush them out to the parallel file system. Further, in order to avoid an I/O storm at large scales, `pio` organizes tasks into I/O groups. Within each group, one writer task performs the actual file I/O on behalf of all other group members. A race condition in the complex writer nomination algorithm—optimized for a platform-specific I/O forwarding constraint—and overlapping consecutive I/O operations causes the intermittent

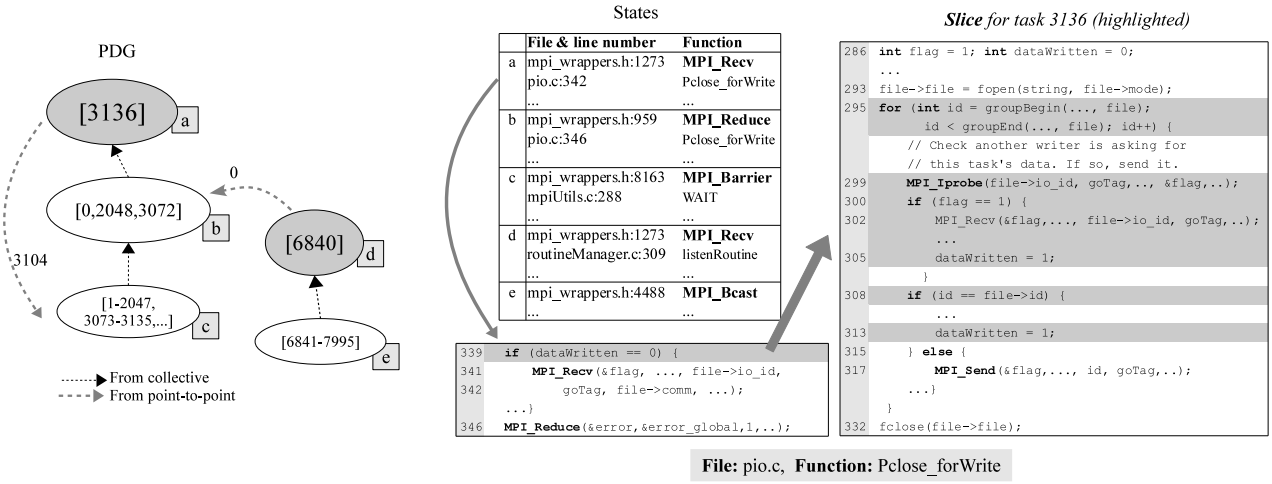

Fig. 5. Output for ddcMD bug.

hang. The application scientist stated that the LP task identification and highlighted statements would have provided him with critical insight about the error. He further verified that a highlighted statement was the bug site.

Further details on the origin of the bug and its fix can be found in the online supplemental material.

5.2 Fault Injection

We inject a local application hang by suspending the execution of a randomly selected process for a long period, which activates our timeout error detection mechanism. We use dynamic binary instrumentation to inject the fault as a sleep call inside randomly selected function calls. Our injector first profiles a run of the application so that we choose from functions that are used during the run. This ensures that all injections result in errors. We only inject in user-level function calls. We do not inject in MPI function calls to reduce the number of experiments—the MPI library has a large number of unique functions and it is often tested better than user applications. We perform all fault-injection experiments on a Linux cluster with nodes that have six 2.8 GHz Intel Xeon processors and 24 GB of RAM. We use 128 tasks in each experiment.

5.2.1 Applications

We inject faults into the NAS Parallel Benchmarks (NPB) [11] (eight benchmarks) and two Sequoia benchmarks: AMG2006 and LAMMPS [12]. The Sequoia benchmarks codes are representative of large-scale HPC production workloads. AMG2006 is a scalable iterative solver for large structured sparse linear systems. LAMMPS is a classical molecular dynamics code. For AMG-2006, we use the default 3D problem (test 1) with the same size in each dimension. For LAMMPS, we use "crack", a crack propagation example in a 2D solid. For the NPBs, we use the class A problem. We limit the number of functions in which faults are injected to 50. NPBs execute fewer than 50 functions so we inject into all functions in the NPBs.

5.2.2 Metrics

Due to its probabilistic nature, *AutomaDeD* may identify LP tasks incorrectly. Also it may identify more than one LP task, which could be correct or incorrect. Multiple LP tasks could be correct when multiple tasks block independently in computation code. In our experiments, since we only inject a fault in one task, we should always have a single LP task.

We use two metrics to evaluate *AutomaDeD*: *accuracy* and *precision*. Accuracy is the fraction of cases in which the set of LP tasks that *AutomaDeD* finds includes the faulty task. Precision is the fraction of cases in which the set of LP tasks that *AutomaDeD* finds includes other (non-faulty) tasks. Accuracy measures the rate of true positives, whereas precision measures the rate of false positives. A false positive impacts users as they could spend time in debugging a task that is not necessarily the LP task. Fig. 6 shows examples of PDGs and LP tasks where a fault is injected in task 3. In case (a), *AutomaDeD* is accurate and precise. In case (b), it is accurate but not precise (since other non-faulty tasks are detected as LP tasks). In case (c), it is neither accurate nor precise; task 3 is actually dependent on other tasks according to *AutomaDeD*. Notice that in case (c), task 3 is the only task in a PDG node (i.e., a singleton task)—we call this

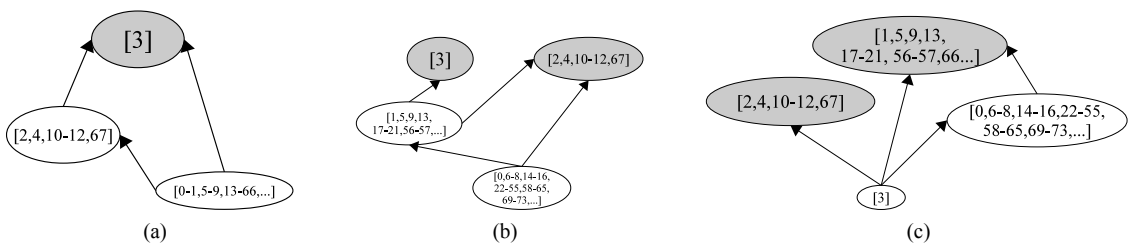

Fig. 6. Examples of PDGs indicating LP tasks (in gray color) for AMG2006. Errors are injected in task 3.

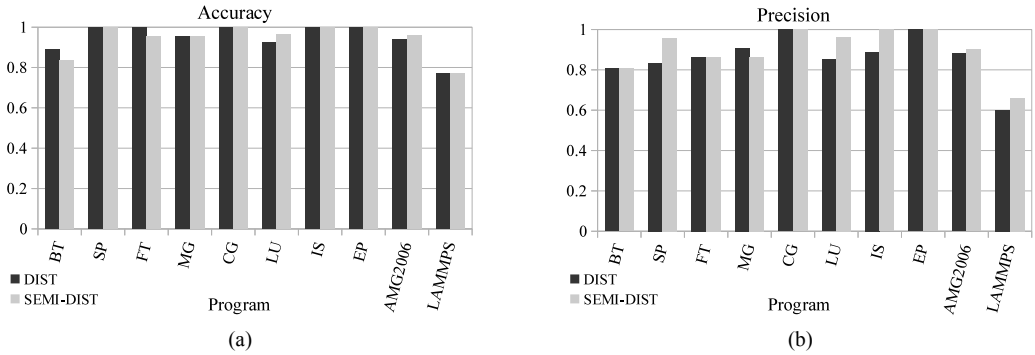

Fig. 7. Accuracy and precision of *AutomaDeD* in detecting the LP task.

isolation. A singleton task appears suspicious to a user so we consider isolation as semi-successful.

5.2.3 Results

Figs. 7a and 7b show the results of the experiments. Overall, *AutomaDeD* has a high accuracy and precision for most benchmarks. Algorithms *DIST* and *SEMI-DIST* have comparable accuracy. However, *SEMI-DIST* has higher or equal precision than *DIST* in nine out of the ten benchmarks. The reason *SEMI-DIST* outperforms *DIST* is due to its improved view of the Markov model—a global MM view—when creating the PDGs which reduces its tendency to infer LP tasks incorrectly. We also notice that, in all the cases when the LP task is inaccurately detected, e.g., case (c) in Fig. 6, the LP task is isolated.

5.3 Performance Evaluation

To evaluate *AutomaDeD*'s performance, we measure slowdown, memory usage, and scalability. *AutomaDeD* has moderate memory usage (to bookkeep MMs) and little slowdown—the worst slowdown was 1.67 for the SP benchmark. Further details on slowdown and memory usage can be found in the online supplemental material.

To measure scalability, we select the benchmark that most stresses *AutomaDeD*'s PDG analysis. The two variables that affect *DIST* and *SEMI-DIST* are the number of tasks—which we vary in the experiments—and the size of the Markov model. We measure the MM sizes for all the benchmarks and select the benchmark that produces the largest MM. Table 3 shows the MM sizes for all the benchmarks. We run all of them with 1,024 parallel task (except for AMG2006 which is run with 1,000 tasks). We select AMG2006, as it produces the largest MM, with 761 edges.

We run AMG2006 with up to 32,768 MPI tasks on an IBM BlueGene/Q machine and measure the time for *AutomaDeD* to perform the distributed part of its analysis using both *DIST* and *SEMI-DIST*. We inject a fault close to its final execution phase in order to have the largest possible MM. Fig. 8 shows the results of the experiments. The analysis time grows logarithmically with respect to the number of tasks as expected. Notice that *SEMI-DIST* takes more time than *DIST*—also as expected—because of the large overhead of reducing the Markov model from all tasks. Our results demonstrate the scalability of *AutomaDeD*. The *DIST* takes less than a second on up to 32,768 MPI tasks.

TABLE 3
Number of Edges of the Markov Model for All the Programs

Program	BT	SP	FT	MG	CG	LU	IS	EP	AMG2006	LAMMPS
MM size	303	263	32	456	136	149	26	17	761	269

6 Related Work

The traditional debugging paradigm [13], [14], [15] of interactively tracking execution of code lines and inspecting program state does not scale to existing high-end systems. Recent efforts have focused on the scalability of tools that realize this paradigm [13], [16]. Ladebug [17] and the PTP debugger [18] also share the same goal. While these efforts enhanced debuggers to handle increased MPI concurrency, root cause identification is still time consuming and manual.

Parallel profiling tools such as Slack [19] and PGPROF [20] identify performance bottlenecks in parallel programs so that runtime can be reduced. These tools provide the time spent in each procedure within (critical) paths while *AutomaDeD* automates the analysis to find the task(s) and code regions in which a performance fault or a correctness problem manifests itself. Unlike *AutomaDeD*, the profilers do not address scalability issues such as the cost of aggregating profiling traces from a large number of MPI tasks.

AutomaDeD is an extension of our previous work [2]. In this paper we present a new algorithm for computing progress dependencies and we evaluate the tool extensively with ten benchmarks. *AutomaDeD*'s root-cause analysis target general coding errors in large-scale scientific codes. Related research work includes probabilistic tools [21], [22],

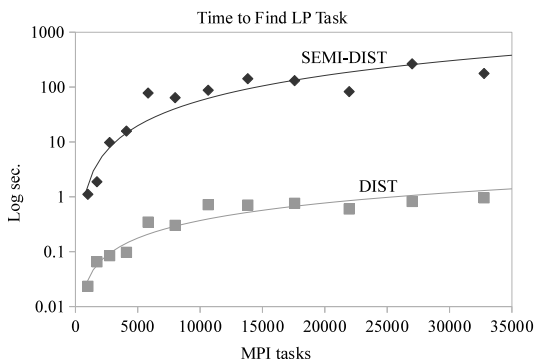

Fig. 8. Time to find LP task in AMG2006.

[23], [24] that detect errors through deviations of application behavior from a model. AutomaDeD [23] and Mirgorodskiy et al. [24] monitor the application's timing behaviors and focus the developer on tasks and code regions that are unusual. Other tools target specific error types, such as memory leaks [25] or MPI coding errors [22], [26], [27], [28]. These tools are complimentary to *AutomaDeD* as they can detect a problem and trigger *AutomaDeD*'s diagnosis.

The closest prior work to *AutomaDeD* is STAT [1], which provides scalable detection of task behavioral equivalence classes based on call stack traces. Its temporal ordering relates tasks by their logical execution order so a developer can identify the least- or most-progressed tasks. However, STAT primarily assists developers in the use of traditional debuggers while *AutomaDeD* detects abnormal conditions and locates the fault automatically.

Others have explored program slicing in MPI programs to locate code sites that may lead to errors. To provide higher accuracy, most techniques use dynamic slicing [7], [8], [9]. These tools tend to incur large runtime overheads and do not scale. Also, techniques must include communication dependencies into data-flow analysis, which is also expensive, to avoid misleading results. *AutomaDeD* uses other information to limit the overhead of slicing.

7 Conclusion

Our novel debugging approach can diagnose faults in large-scale parallel applications. By compressing historic control-flow behavior of MPI tasks using Markov models, our technique can identify the least progressed task of a parallel program by inferring probabilistically a progress-dependence graph. We use backward slicing to pinpoint code that could have led to the unsafe state. We design and implement *AutomaDeD*, which diagnoses the most significant root-cause of a problem. Our analysis of a hard-to-diagnose bug and fault injections in three representative large-scale HPC applications demonstrate that *AutomaDeD* identifies these problems with high accuracy, where manual analysis and traditional debugging tools have been unsuccessful. The distributed part of the analysis can be performed in a fraction of a second with over 32 thousand tasks. The low analysis cost allows its use multiple times during program execution.

Acknowledgments

The authors thank David Richards of the Lawrence Livermore National Laboratory for helping us to conduct the blind study on ddcMD. This work was partly supported by the National Science Foundation under Grant No. CNS-0916337, and it was performed partly under the auspices of the US Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-JRNL-643939).

References

[1] 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 *Proc. Conf. High Performance Comput. Netw., Storage Anal.*, 2009, pp. 1–11.

[2] 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 *Proc. 21st Int. Conf. Parallel Archit. Compilation Tech.*, 2012, pp. 213–222.

[3] F. H. Streitz, J. N. Glosli, M. V. Patel, B. Chan, R. K. Yates, B. R. de Supinski, J. Sexton, and J.A. Gunnels, "Simulating solidification in metals at high pressure: The drive to petascale computing," in *J. Phys.: Conf. Ser.*, vol. 46, no. 1, pp. 254-267, 2006.

[4] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Performance Comput. Appl.*, vol. 19, pp. 49–66, 2005.

[5] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449.

[6] B. Korel and J. Laski, "Dynamic slicing of computer programs," *J. Syst. Softw.*, vol. 13, no. 3, pp. 187–195, Dec. 1990.

[7] M. Kamkar, P. Krajina, and P. Fritzson, "Dynamic slicing of parallel message-passing programs," in *Proc. 4th Euromicro Workshop Parallel Distrib. Process.*, Jan. 1996, pp. 170–177.

[8] J. Rilling, H. Li, and D. Goswami, "Predicate-based dynamic slicing of message passing programs," in *Proc. IEEE 2nd Int. Workshop Source Code Anal. Manipulation*, 2002, pp. 133–142.

[9] G. Shanmuganathan, K. Zhang, E. Wong, and Y. Qi, "Analyzing message-passing programs through visual slicing," in *Proc. Int. Conf. Inf. Technol. Coding and Comput.*, vol. 2, Apr. 2005, pp. 341–346.

[10] M. Strout, B. Kreaseck, and P. Hovland, "Data-flow analysis for MPI programs," in *Proc. Int. Conf. Parallel Process.*, Aug. 2006, pp. 175–184.

[11] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, Mountain View, CA, USA, Rep. RNR-91-002, Aug. 1991.

[12] ASC Sequoia Benchmark Codes, (2013). [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks/.

[13] Allinea Software Ltd, "Allinea DDT—Debugging tool for parallel computing," (2013). [Online]. Available: http://www.allinea.com/products/ddt/

[14] GDB Steering Committee, "GDB: The GNU Project Debugger," (2013). [Online]. Available: http://www.gnu.org/software/gdb/documentation/

[15] Rogue Wave Software, "TotalView Debugger," (2013). [Online]. Available: http://www.roguewave.com/products/totalview.aspx.

[16] J. DelSignore. (2003, Oct.) "TotalView on Blue Gene/L," Presented at "Blue Gene/L: Applications, Architecture and Software Workshop", Oct. 2003. [Online]. Available: https://asc.llnl.gov/computing_resources/bluegenel/papers/delsignore.pdf.

[17] S. M. Balle, B. R. Brett, C. Chen, and D. LaFrance-Linden, "Extending a traditional debugger to debug massively parallel applications," *J. Parallel Distrib. Comput.*, vol. 64, no. 5, pp. 617–628, 2004.

[18] G. Watson and N. DeBardeleben, "Developing scientific applications using eclipse," *Comput. Sci. Eng.*, vol. 8, no. 4, pp. 50–61, 2006.

[19] J. Hollingsworth and B. Miller, "Parallel program performance metrics: A comparison and validation," in *Proc. Supercomput.*, Nov. 1992, pp. 4–13.

[20] The Portland Group, "PGPROF graphical performance profiler," (2013). [Online]. Available: http://www.pgroup.com/products/pgprof.htm

[21] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. Ahn, and M. Schulz, "AutomaDeD: Automata-based debugging for dissimilar parallel tasks," in *Proc. IEEE/IFIP Conf. Dependable Syst. Netw.*, 2010, pp. 231–240.

[22] Q. Gao, F. Qin, and D. K. Panda, "DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *Proc. ACM/IEEE Supercomput. Conf.*, 2007, pp. 15:1–15:12.

[23] 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 *Proc. ACM/IEEE Supercomput. Conf.*, 2011, pp. 50:1–50:10.

[24] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *ACM/IEEE Supercomput Conf.*,, New York, NY, USA: ACM, 2006, p 11.

[25] S. C. Gupta and G. Sreenivasamurthy, "Navigating C̈in a L̈eakyB̈oat? Try purify," *IBM developerWorks*, 2006. [Online]. Available: www.ibm.com/developerworks/rational/library/06/0822_satish-giridhar/

[26] Q. Gao, W. Zhang, and F. Qin, "FlowChecker: Detecting bugs in MPI libraries via message flow checking," in *Proc. ACM/IEEE Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2010, pp. 1–11.
[27] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A graph based approach for MPI deadlock detection," in *Proc. Int. Conf. Supercomput*, 2009, pp. 296–305.
[28] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with umpire," in *Proc. ACM/IEEE Supercomput. Conf.*, 2000, pp. 51:1–51:10.
[29] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.
[30] M. Kamkar and P. Krajina, "Dynamic slicing of distributed programs," in *Proc. Int. Conf. Softw. Maintenance*, Oct. 1995, pp. 222–229.

Ignacio Laguna received the BS degree in electronics and communications engineering from Universidad de Panamá in 2002, and the MS and PhD degrees from the School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana, in 2008 and 2012, respectively. He is currently a postdoctoral researcher at the Lawrence Livermore National Laboratory. He received the ACM and IEEE CS George Michael Memorial Fellowship in 2011 for his work on large-scale failure diagnosis techniques. His research interests include software reliability, fault tolerance, and anomaly detection in high-performance computing. He is a member of the IEEE.

Dong H. Ahn received the master's degree in computer science from the University of Illinois at Urbana Champaign in 2001. He has been working for the Development Environment Group at Lawrence Livermore National Laboratory since 2001. During this period, he has worked on several code-development-tools projects with a common goal to provide highly capable and scalable tools ecosystems for large computing systems. Toward this goal, he has negotiated and managed key software development contracts to improve the capabilities and scalability of the TotalView parallel debugger, architected LLNL's extreme-scale debugging strategy that conceived the Stack Trace Analysis Tool (STAT). He is a 2011 R&D 100 Award winner, and has invented novel software infrastructure technologies as well as led interactions with vendors to enable this strategy. His current interest includes scalable approaches to massively scalable dynamic loading, to managing the adverse impacts of nondeterminism in concurrent execution, and to next-generation resource management. Prior to joining LLNL, he worked for the National Center for Supercomputing Applications (NCSA). He is a member of the IEEE Computer Society and the ACM.

Bronis R. de Supinski received the PhD degree in computer science from the University of Virginia in 1998 and joined the Center for Applied Scientific Computing (CASC) in July 1998. He is the chief technology officer (CTO) for Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL). In this role, he is responsible for formulating LLNL's large-scale computing strategy and overseeing its implementation. His position requires frequent interaction with high-performance computing (HPC) leaders and he oversees several collaborations with the HPC industry as well as academia. Prior to becoming the CTO for LC, he led several research projects in LLNL's Center for Applied Scientific Computing. Most recently, he led the Exascale Computing Technologies (ExaCT) project and co-led the Advanced Scientific Computing (ASC) program's Application Development Environment and Performance Team (ADEPT). ADEPT is responsible for the development environment, including compilers, tools, and run-time systems, on LLNL's large-scale systems. ExaCT explored several critical directions related to programming models, algorithms, performance, code correctness, and resilience for future large-scale systems. He currently continues his interests in these topics, particularly programming models, and serves as the chair of the OpenMP Language Committee. In addition to his work with LLNL, he is also a professor of exascale computing at the Queen's University of Belfast and an adjunct associate professor in the Department of Computer Science and Engineering at Texas A&M University. Throughout his career, he has won several awards, including the prestigious Gordon Bell Prize in 2005 and 2006, as well as an R&D 100 for his leadership of a team that developed a novel scalable debugging tool. He is a member of the IEEE Computer Society and the ACM.

Saurabh Bagchi is a professor in the School of Electrical and Computer Engineering and the Department of Computer Science (by courtesy) at Purdue University in West Lafayette, Indiana. He is a distinguished scientist of the ACM, a distinguished speaker for the ACM, an IMPACT faculty fellow at Purdue (2013-14), and an assistant director of the CERIAS security center at Purdue. He leads the Dependable Computing Systems Laboratory (DCSL), where his group performs research in practical system design and implementation of dependable distributed systems. Since 2011, he has been serving as a visiting scientist with IBM Research. He is a senior member of IEEE and the ACM.

Todd Gamblin received the BA degree in computer science and Japanese from Williams College in 2002, and the MS and PhD degrees in computer science from the University of North Carolina at Chapel Hill in 2005 and 2009, respectively. He is a computer scientist in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). His research focuses mainly on scalable algorithms for measuring, analyzing, and visualizing performance data from massively parallel applications. He is also interested in fault tolerance, resilience, MPI, and parallel programming models. He has been at LLNL since 2008. He works closely with researchers in CASC and with staff in the Development Environment Group in Livermore Computing. He is the team leader for the performance analysis and visualization at Exascale (PAVE) project, and he also works on the Exascale Computing Technologies LDRD project, the SciDAC Sustained Performance, Energy, and Resilience (SUPER) project, and many other ASC projects at LLNL. He has also worked as a software developer in Tokyo and held graduate research internships at the University of Tokyo and IBM Research. He is a member of the IEEE.