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

Journal:	*Transactions on Parallel and Distributed Systems*
Manuscript ID:	TPDS-2013-09-0903
Manuscript Type:	Regular
Keywords:	D.2.5 Testing and Debugging < D.2 Software Engineering < D Software/Software Engineering, D.2.5.b Debugging aids < D.2.5 Testing and Debugging < D.2 Software Engineering < D Software/Software Engineering, D.2.5.c Diagnostics < D.2.5 Testing and Debugging < D.2 Software Engineering < D Software/Software Engineering, D.2.5.d Distributed debugging < D.2.5 Testing and Debugging < D.2 Software Engineering < D Software/Software Engineering, G.1.0.g Parallel algorithms < G.1.0 General < G.1 Numerical Analysis < G Mathematics of Computing

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

Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, Todd Gamblin

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 ten HPC benchmarks and show that our analysis takes less than a few seconds on thousands of parallel tasks.

Index Terms—Distributed debugging, diagnostics, parallel systems

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, for example, registers and program variables, 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 different a code region or on a different process from their original cause. Often, when a parallel MPI application suffers from a performance fault, the manifestation of the fault may occur in separate tasks from its root cause. For example, consider a faulty task that deadlocks in computation code in an early phase of the application. As a consequence, non-faulty tasks may block later—and cannot make further progress—if they execute collective or blocking point-to-point communication operations that involve the faulty task. To provide insight into these problems, previous work [1], [2] identifies the *least-progressed (LP)* task (or tasks) in 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 (possibly thousands) of tasks to one (or a few) task(s).

In contrast to STAT [1], which identifies the LP task via temporal ordering using static and dynamic analysis, the *AutomaDeD* (Automata-based debugging for dissimilar parallel tasks) tool identifies it via low-cost dynamic analysis [2]. *AutomaDeD* probabilistically identifies the LP task (or tasks) by using a Markov Model (MM) as a lightweight, statistical summary of each task's control-flow history. States in the Markov model represent MPI calls and regions of computation in application code, and edges represent state transitions. The models are created online with little runtime overhead and provide rich information that allows *AutomaDeD* to trace 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. We use static program slicing [3] as an example of the code-analysis techniques that can be applied scalably in our framework.

- *I. Laguna, D. H. Ahn, B. R. de Supinski and T. Gamblin are with the Lawrence Livermore National Laboratory, Livermore, California, 94550. E-mail: {ilaguna, ahn1, bronis, tgamblin}@llnl.gov*
- *S. Bagchi is with Purdue University, West Lafayette, Indiana, 47907.*

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. MM may be different among tasks (i.e., they may have different states and edges) due to control flow differences. 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 among MM could cause impression when computing progress dependencies since dependencies could conflict to each other (e.g., in one task the dependence is one direction, whereas in another task, the dependence is in the opposite direction), and could be null locally but non-null globally (i.e., a dependence may not be inferred only from local information, but it can be inferred from globally aggregated information).

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 they incur only slight runtime overhead. Our implementation is transparent—it uses the MPI profiling interface to intercept communication calls—and it does not require separate daemons to trace the application as other debugging tools do (e.g., TotalView [4] and STAT [1]).

The previous fault coverage evaluation of *DIST* (i.e., measurements of accuracy and precision) only included two of the Sequoia benchmarks: AMG2006 and LAMMPS [5]. This paper presents a fault coverage evaluation of *DIST* and *SEMI-DIST* with the NAS Parallel Benchmarks [6] in addition to AMG2006 and LAMMPS—a total of ten benchmarks. This allows us to measure and understand the efficiency of our techniques with a larger variety of HPC codes.

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% of the cases (using *DIST*), averaged across the tested benchmarks. Its precision (i.e., the inverse of false-positive rate) is 90% 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 [7] 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:

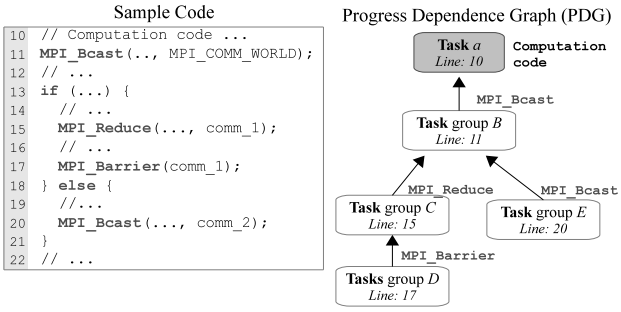

Fig. 1: Progress dependence graph example. Task a blocks in computation code which leads to groups of tasks B, C, D and E to block in other code regions, i.e., collective MPI calls. Task groups cannot progress due to task a.

- A 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 ten HPC benchmarks;
- A study of scalability *DIST* and *SEMI-DIST* with up to 32,768 MPI tasks;
- A detailed description of a blind study, which demonstrates the effectiveness of *AutomaDeD* in pinpointing the root cause of an elusive concurrency bug in a molecular dynamics application.

The rest of the 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* (PDG) 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, which can subsequently aid in performance tuning the application (e.g., by highlighting tasks with the least progress).

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 [8]. 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

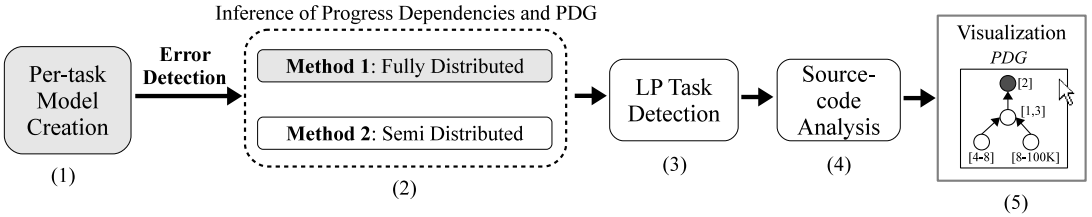

Fig. 2: Diagnosis work flow (gray blocks execute in a distributed fashion—a single task performs the others).

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 $Y \xrightarrow{pd} Z$.*

Figure 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. We formalize the progress-dependence for point-to-point operations as follows:

Definition 2 (Point-to-Point Progress Dependence)**.** *If task x blocks when sending (receiving) a message to (from) task y at time t, then $x \xrightarrow{pd} y$.*

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 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 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 Figure 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 (LP) 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 Figure 1, the PDG shows where to focus attention, i.e., the LP task(s). Thus, debugging time is substantially reduced, as the programmer focuses on the execution contexts of a few tasks, instead of potentially thousands of tasks in a large-scale run. Second, we can efficiently apply static or dynamic bug-detection methods based on the state of LP task(s). *AutomaDeD* applies slicing [3] starting from the state of the LP task, which substantially reduces the search space of slicing when compared to slicing the execution context of each task (or representative task group) separately and then combining this information to find the origin of the fault.

2.1.3 PDG Versus Other Dependency Graphs

A PDG is different from the dependency graph used in prior work on MPI deadlock detection [9], [10], [11]. A PDG hierarchically describes the execution progress of MPI tasks. It addresses questions such as: Which task has made the least progress? Which tasks does the LP task prevent from making progress? In contrast, knots in traditional dependency graphs can detect real and potential deadlocks. We do not detect deadlocks by checking for knots in a PDG. However, since a PDG combines dependencies arising from MPI operations, it can indicate that a deadlock caused a hang. Performance problems are a superset of hangs; deadlocks or other causes can lead to hangs. Our case study with a real-world bug in Section 5.1 shows an example in which we use a PDG to identify a deadlock as the root-cause of a hang.

2.2 Workflow of Our Approach

We have developed a tool called *AutomaDeD* to diagnose failures using a PDG. Figure 2 shows the steps of this diagnosis. Steps 1–2 are distributed in *DIST* and semi-distributed in *SEMI-DIST*, whereas steps 3–5 are performed in a single task (in both algorithms).

1) **Markov-model creation:** *AutomaDeD* captures per-MPI-task control-flow behavior in a Markov model (MM). 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. Our previous work uses similar Markov-like models (semi-Markov models) to find similarities between tasks to detect errors [12], [13]. Our novel framework uses MMs to summarize control-flow execution history as Section 3.1 details. No prior work, to our knowledge, uses MMs to infer progress dependencies.
2) **Inferring dependencies and PDG creation:** When a system (either *AutomaDeD* or a third-party system) detects a performance fault, *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. This algorithm uses an all-reduce over the MM state of each task, which provides each task with the state of all other tasks. Formally, if a task's local state is s_{local}, this operation provides each task with the set $S_{\text{others}} = s_1, \ldots, s_j, \ldots, s_N$, where $s_j \neq s_{\text{local}}$. Next, each task probabilistically infers its own local PDG based on s_{local} and S_{others}. Finally, it reduces these PDGs to a single 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 it has almost the same accuracy; *SEMI-DIST* can eliminate conflicting dependencies that *DIST* cannot since it uses a global MM as input. However, creating a global MM to compute a PDG incurs extra overhead as a global reduction is performed to merge all the MM in one.
3) **LP-task detection:** Based on the reduced PDG, we determine the LP task and its state (i.e., call stack and program counter), which we use in the next step (i.e., source-code analysis). The LP task(s) are a group of tasks without any progress dependence. From the perspective of a PDG, *AutomaDeD* only has to find nodes with no outgoing edges. Since this is not an expensive operation—most of the PDGs in our experiments had between 5 to 10 nodes—it can be executed sequentially without substantial overhead.
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. Examples of applicable source-code analysis are data-flow analysis, points-to analysis, and program slicing. 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. We perform slicing in *AutomaDeD* using Dyninst [14].
5) **Visualization:** Finally, *AutomaDeD* presents the program slice, the reduced PDG and its associated information. The user can attach a serial or parallel debugger to the LP task identified in the PDG. The PDG also provides other task groups and their dependencies. The slice brings the programmer's attention to the code that affected the LP task, and allows them to find the fault.

3 Design

In this section, we describe a statistical technique to model the control-flow of parallel tasks as a Markov model. We then present the inference of progress dependence based on this model.

3.1 Summarizing Execution History

A simple, direct approach to save the control-flow execution history is to build a control-flow graph (CFG) based on executed statements [15]. This approach typically requires heavy-weight (binary or source-code) instrumentation and it is only feasible for small applications. Since large-scale MPI applications can have very large CFGs, *AutomaDeD* instead captures a compressed version of the control-flow behavior using our MM with communication and computation states. The edge weights capture the frequency of transitions between two states. Figure 3 shows how *AutomaDeD* creates MMs at runtime in each task. We use the MPI profiling interface to intercept MPI routines. Before and after calling the corresponding PMPI routine, *AutomaDeD* captures stateful information, such as a call stack trace, offset address within each active function, and return address. We assume that the MPI program is compiled using debugging information so that we can resolve addresses to line numbers in source files.

Note that each task computes its own version of a Markov model, which is based only on the states that it has seen. MMs across different tasks may be different—they may have different states and edges, and similar edges between two tasks could have different transition probabilities. This reflects the SPMD (single program, multiple data) nature of MPI programs and our distributed approach to build MMs—tasks do not synchronize when creating MMs.

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

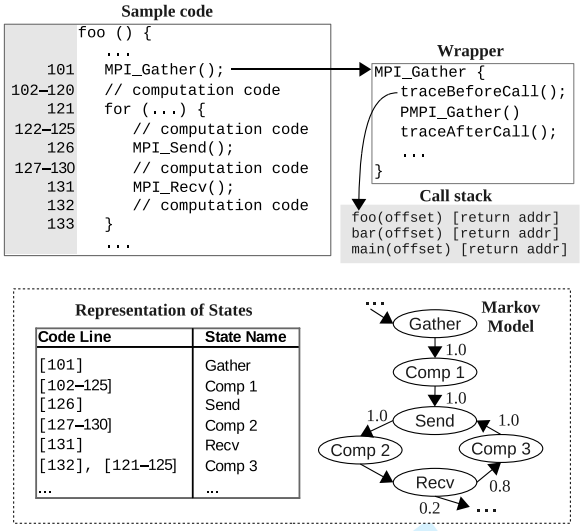

Fig. 3: Markov model creation. States represent code regions within or between MPI calls. Edge annotations represents the frequency of transitioning from one state to another state.

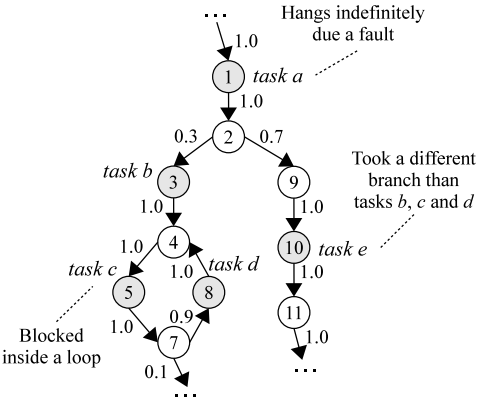

Fig. 4: Sample Markov model with five tasks that blocked in different states

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. In cases 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 for `MPI_ANY_SOURCE`, and in this case, *AutomaDeD* simply omits this progress-dependence edge.

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., $P(y, x)$.*

Figure 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 Figure 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 task b and task c, we calculate that 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 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 Figure 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. They are in different execution branches so the forward and

TABLE 1: Dependence based on path probabilities. Undefined dependencies are denoted as "?."

$P(i,j)$			$P(j,i)$			Dependence?	Type
0	$0<P<1$	1	0	$0<P<1$	1		
✓			✓			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	$t_i \xleftarrow{pd} t_j$
		✓			✓	?	

backward path probabilities between their states, i.e., $P(3,10)$ and $P(10,3)$, are both zero. 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 $t_j \xrightarrow{pd} t_i$ since execution goes 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 $t_i \xrightarrow{pd} t_j$ 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.

3.2.3 Procedure to construct PDG locally

Algorithm 1 shows our local PDG construction procedure. It is a *local* procedure since it is performed by a single task after global information is gathered.

Algorithm 1 Computes PDG (locally for a task)

Input: mm: Markov model
$closure$: transitive closure of the Markov model
$statesSet$: set of current states of all tasks
Output: $matrix$: adjacency-matrix representation of PDG

```
1:  procedure LocalPDG
2:      currState ← current state of the task (in mm)
3:      for all state in statesSet do
4:          if state ≠ currState then
5:              f ← PathProbability(currState, state)
6:              b ← PathProbability(state, currState)
7:              dep ← dependence based on f, b
8:              matrix[currState, state] ← dep
9:          end if
10:     end for
11: end procedure
12:
13: function PathProbability(src, dst)
14:     p ← 0
15:     if src can reach dst in closure then
16:         for all path between src and dst do
17:             p ← p + probability(path, src, dst)
18:         end for
19:     end if
20:     return p
21: end function
```

Section 4 describes two algorithms (named, *DIST* and *SEMI-DIST*) to collect the input global-information for this procedure. Algorithm 1 requires as input an MM, a transitive closure of the MM, and $statesSet$, the states of all other tasks. We compute the dependencies between the current state and all the states of $statesSet$. We represent dependencies as integers (0: no dependence; 1: forward dependence; 2: backward dependence; 3: undefined). We save the PDG in an adjacency matrix. Line 7 determines dependencies based on all-path probabilities and Table 1. The overall complexity of this algorithm is $O(s\times(|V|+|E|))$, where s is the number of states in $statesSet$, and $|V|$ and $|E|$ are the numbers of states and edges of the MM. In practice, MMs are sparse graphs in which $|E| \approx |V|$, so time complexity is approximately $O(s \times |E|)$.

3.2.4 Comparison to postdominance.

Our definition of progress dependence is similar to the concept of postdominance [16] in which a node j of a CFG postdominates a node i if every path from i to the *exit* node includes j. However, our definition does not require the *exit* node to be in the MM—postdominance algorithms require it to create a postdominator tree. Since a fault could cause the program to stop in any state, we are not guaranteed to have an exit node within a loop. Techniques such as assigning a dummy exit node to a loop do not work in

general for fault diagnosis because a faulty execution makes it difficult (or impossible) to determine the right exit node. To use the postdominance theory, we could use static analysis to find the exit node and map it to a state in the MM. However, our dynamic analysis approach is more robust and should provide greater scalability and performance.

4 Scalable Mechanisms

This section details our PDG analysis implementation that ensures scalability with increasing numbers of MPI tasks. In particular, we focus on two algorithms that can build a PDG. The first algorithm (*DIST*) is fully distributed—i.e., all tasks create their own PDGs that are subsequently reduced down to a single PDG at the root task. The second algorithm (*SEMI-DIST*) first designates a task to create a global Markov model (by globally reducing all edges and transition probabilities) and then builds the global PDG. We will examine the performance and precision trade-offs of these schemes in Section 5. *AutomaDeD* is implemented in C++ and uses the Boost Graph Library [17] for graph-related algorithms such as depth-first search and for the construction of transitive closures in the Markov Model.

4.1 Monitoring Mechanisms

4.1.1 Error Detection

Before triggering the PDG-based analysis, we assume that a performance problem has been detected: e.g., the application is not producing its output within an expected time frame. The user can then use *AutomaDeD* to find the tasks and the associated code region that caused the problem. *AutomaDeD* includes a timeout detection mechanism that can trigger the diagnosis analysis, and it can infer a reasonable timeout threshold, i.e., based on the mean time and standard deviation of state transitions). The user can also supply the timeout as an input parameter. Our experiments with large-scale HPC applications found that a 60-second threshold is sufficient.

4.1.2 Helper thread

AutomaDeD uses a helper thread to analyze the MM and performs the core of the dependence inference: Step 2 in Figure 2. Unlike other Steps in which only one task (MPI rank 0 by default) performs inexpensive operations, step 2 is required to be distributed (or semi-distributed) and scalable. Thus, *AutomaDeD* uses `MPI_THREAD_MULTIPLE` to initialize MPI and enable the helper threads to use MPI. On machines that do not support threads, such as Blue Gene/L, we save all MMs to the parallel file system when we detect an error. *AutomaDeD* then reads these MMs for analysis in a separate MPI program.

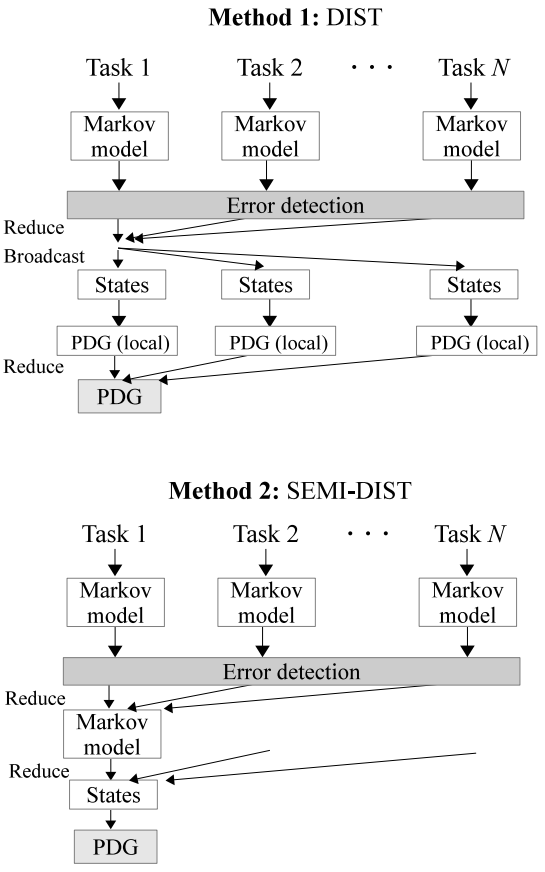

Fig. 5: Workflow of the algorithms build the PDG

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

4.2 Inference of the PDG

4.2.1 Distributed algorithm

Algorithm 2 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 1 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). Figure 5 illustrates this algorithm.

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`

Algorithm 2 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
```

since we represent dependencies as integers.

We cannot use `MPI_Reduce` for our reduction steps since tasks can contribute states of different sizes so we implement custom reductions that use binomial trees. These operations have $O(\log p)$ complexity where p is the number of tasks. Assuming a scalable broadcast implementation, the overall complexity is also $O(\log p)$. Our algorithm can therefore scale to the task counts found on even the largest HPC systems.

4.2.2 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 (i.e., locally on each task). Figure 5 illustrates the workflow of this algorithm.

Algorithm 3 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 of *SEMI-DIST* is also $O(\log p)$. However, this algorithm generally incurs higher overall overhead than *DIST*. The most expensive step of *DIST* is the reduction of the PDGs while that in *SEMI-DIST* is the reduction of the MMs. In general, PDGs tend to have much fewer numbers of edges than MMs and hence reducing MMs is more expensive than PDGs. The reduction of states is common in both of them and this is relatively inexpensive. In our experiments, the size of PDGs is in the range of 5–10 edges while the size of MMs varies between 200–1,000 edges. Thus, *SEMI-DIST* is a trade-off that exchanges the performance with better precision, as we show in our experimental evaluation in Section 5.

4.3 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).

4.4 Guided Application of Program Slicing

4.4.1 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 [3], which is valid for all possible executions. Dynamic slicing [18] 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 [19], [20], [21]. 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 [20], [21]. 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 [22]. 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. Statements 4–5 should be in the slice because the value `x` sent is received as `y`, which obviously 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 [22] to construct a slice based on a particular statement or variable. Further, the MPI developer must decide on which tasks to apply communication-aware static slicing since applying it to every task is infeasible at large scales.

4.4.2 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.

This slicing method reduces the complexity of manually applying static slicing to diagnose a failure. The user can simply start with the most important slice (i.e., the one associated with the LP task) and progressively augment it by clicking the "next" button in a graphical interface, until the fault is found.

5 Evaluation

We demonstrate how *AutomaDeD* diagnoses a difficult bug in a molecular dynamics program that manifested only at large scales. We also evaluate *AutomaDeD* extensively in a controlled setting by performing fault injection in ten MPI benchmarks. We also measure the memory consumption and performance overhead (including scalability) of the tool.

5.1 Case Study

An application scientist challenged us to locate an elusive error in ddcMD, a parallel molecular dynamics code [7]. 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.

Figure 6 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 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.

More specifically, on Blue Gene/L, a number of compute nodes perform their file I/O through a dedicated I/O node (ION) so `pio` nominates *only one* writer task per ION. Thus, depending on how MPI tasks map to the underlying IONs, an I/O group does not always contain its writer task. In this case, `pio` nominates a non-member task that belongs to a different I/O group. This mechanism led to a condition

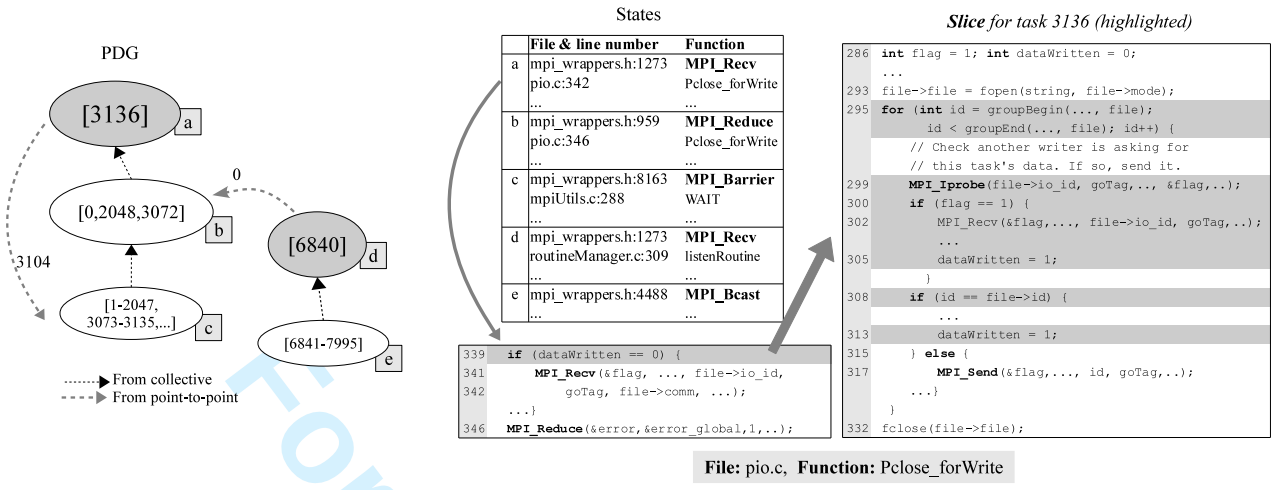

Fig. 6: Output for ddcMD bug.

in which a task plays dual roles: a non-writer for its own I/O group and the writer for a different group.

Figure 6 shows the main loop of a writer. To receive the file buffer from a non-writer, the group writer first sends a request to each of its group members to send the file buffer via the `MPI_Send` at line 317. The group member receives that request via the `MPI_Recv` at line 341 and sends back the buffer size and the buffer. As shown in the loop, a dual-purpose task has an extra logic: it uses `MPI_Iprobe` to test whether it must reply to its non-writer duty while it performs its writer duty. The logic is introduced in part to improve performance. However, completing that non-writer duty frees its associated writer task to move on from MPI blocking communications. The hang arises when two independent instances of `pio` are simultaneously processing two separate sets of buffers. This pattern occurs in the application when a small data set is written immediately after a large data set. Some tasks can still be performing communication for a large data set while others work on a small set. Because the MPI send/recv operations use tags that are fixed at compile time, messages from a small set could be confused for those for a large set of `pio` and vice-versa, leading to a condition in which a task could hang waiting for a message that was intercepted by a wrong instance.

This error only arose on this particular platform because the dual-purpose condition only occurs under Blue Gene's unique I/O forwarding structure. We also theorize that the error emerges only at large scales because this scale increases the probability that the dual-purpose assignments and simultaneous `pio` instances occur. The application scientist had corrected the error through unique MPI tags in order to isolate one `pio` instance from another.

5.2 Fault Injection

In this section, we evaluate the code coverage of *AutomaDeD*'s LP task detection. 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 PIN [23] to inject the fault as a sleep call at the beginning of randomly selected function calls. Our injector first profiles a run of the application so that we randomly 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 so it has fewer bugs. 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) [6] (eight benchmarks) and two Sequoia benchmarks: AMG2006 and LAMMPS [5]. 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

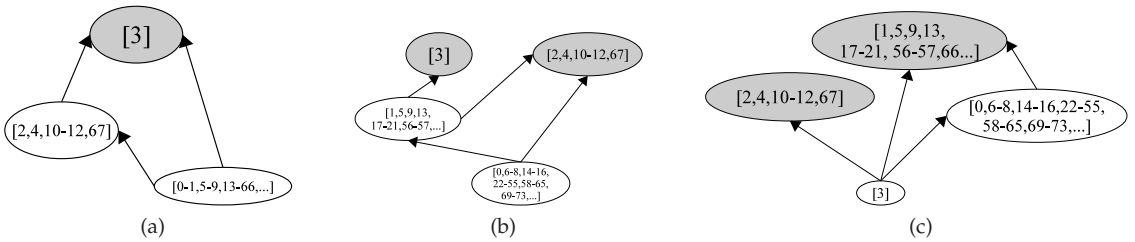

Fig. 7: Examples of PDGs indicating LP tasks (in gray color) for AMG2006. Errors are injected in task 3.

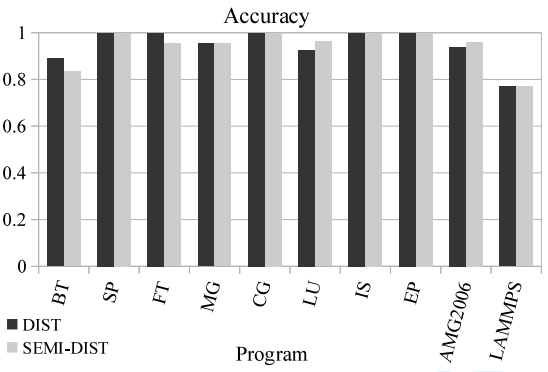

Fig. 8: Accuracy of *AutomaDeD* in detecting the LP task.

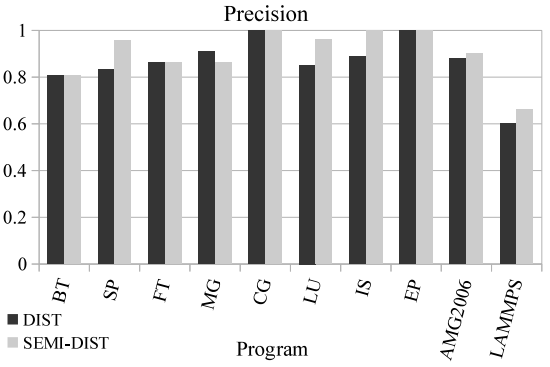

Fig. 9: Precision of *AutomaDeD* in detecting the LP task.

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. Figure 7 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 *isolation*. A singleton task appears suspicious to a user so we consider isolation as semi-successful.

5.2.3 Results

Figures 8 and 9 show the results of the experiments. Overall, *AutomaDeD* has a high accuracy and precision for most of the 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 Figure 7, the LP task is isolated.

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

5.3 Performance Evaluation

5.3.1 Scalability

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. Figure 10

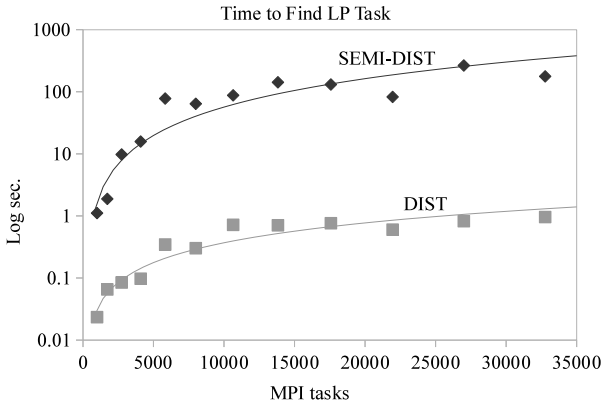

Fig. 10: Time to find LP task in AMG2006.

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. The low cost of this analysis suggests that we can trigger it at multiple execution points with minimal impact on the application run. *SEMI-DIST* is suggested to be used at moderate scales or when waiting a few minutes is not an inconvenience for users.

5.3.2 Slowdown and Memory Usage

Figures 11 and 12 show application slowdown and *AutomaDeD* memory usage for AMG2006, LAMMPS, and the NPBs. Slowdown is the ratio of the application run time with *AutomaDeD* to the run time without it. Memory usage shows the proportional increase in maximum resident set size (RSS) when we use *AutomaDeD* (i.e., *increase = (memory-use-with-tool)/(memory-use-without-tool)*). Since *AutomaDeD* operates as a library, its memory usage increases the memory usage of the tasks themselves. Since tasks can have different memory usage (depending on their behavior), we used the task with the highest memory usage to calculate the increase.

AutomaDeD incurs little slowdown—the worst is 1.67 for SP—because the overhead is primarily the cost of intercepting MPI calls and updating the MM, steps that we have highly optimized. For example, to optimize MM creation, we use efficient C++ data structures and algorithms such as associative containers and use pointer comparisons (rather than string-based comparisons) to compare states. Memory usage is moderate for most benchmarks; the largest is AMG2006 (7.02), which has many (unique) states in its execution. The factor that most affects slowdown is the number of MPI calls from different contexts since this increases the number of states that *AutomaDeD*

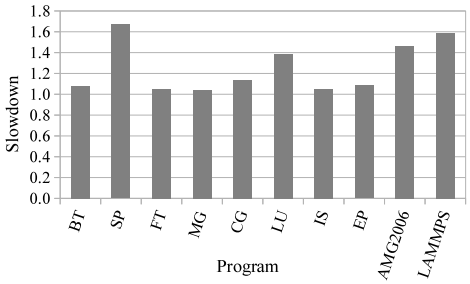

Fig. 11: Application slowdown.

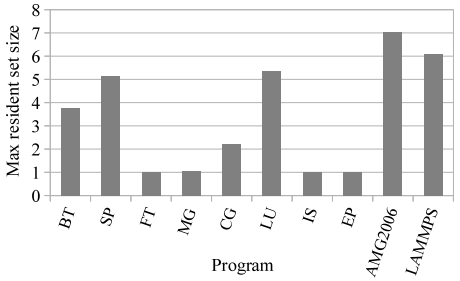

Fig. 12: Increase in maximum memory usage.

creates. Benchmarks with slowdown of 1.3 (or more) call MPI routines from different contexts more often than the others. Applications with higher slowdown (AMG2006, LAMMPS, SP and LU) also exhibit higher memory usage due to the number of states.

6 Related Work

The traditional debugging paradigm [24], [25], [4] 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 [24], [26]. Ladebug [27] and the PTP debugger [28] 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 [29] and PGPROF [30] 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 [12], [31], [13], [32] that detect errors through deviations of application

behavior from a model. AutomaDeD [13] and Mirgorodskiy et al. [32] 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 [33] or MPI coding errors [31], [34], [9], [10], [11]. These tools are complimentary to *AutomaDeD* as they can detect a problem and trigger *AutomaDeD*'s diagnosis.

Assertion-based debugging also targets reduced manual effort. Recent work addresses scalability of parallel assertion-based debugging [35] but does not suit localization of performance faults. Differential debugging [36] provides a semi-automated approach to understand programming errors; it dynamically compares correct and incorrect runs. While these techniques have been applied at small scales [37], the time and scale expenses are prohibitive at large scales.

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 [19], [20], [21]. 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 and to validate the results. 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 U.S. Department of Energy 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 *SC '09*, 2009.
[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 *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12, 2012, pp. 213–222.
[3] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, 1981, pp. 439–449.
[4] Rogue Wave Software, "TotalView Debugger," http://www.roguewave.com/products/totalview.aspx.
[5] "ASC Sequoia Benchmark Codes," https://asc.llnl.gov/sequoia/benchmarks/.
[6] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, RNR-91-002, Aug. 1991.
[7] 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," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 254, 2006.
[8] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.
[9] W. Haque, "Concurrent deadlock detection in parallel programs," *International Journal of Computers and Applications*, vol. 28, pp. 19–25, January 2006.
[10] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A graph based approach for mpi deadlock detection," in *International conference on Supercomputing (ICS)*, 2009, pp. 296–305.
[11] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of mpi applications with umpire," in *ACM/IEEE Supercomputing Conference (SC)*, 2000.
[12] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. Ahn, and M. Schulz, "AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks," in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 231 –240.
[13] 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 *ACM/IEEE Supercomputing Conference (SC)*, 2011, pp. 50:1–50:10.
[14] "DynInst - An Application Program Interface (API) for Runtime Code Generation," http://www.dyninst.org/.
[15] M. Kamkar and P. Krajina, "Dynamic slicing of distributed programs," in *International Conference on Software Maintenance*, oct 1995, pp. 222 –229.
[16] 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.
[17] "Boost C++ libraries," http://www.boost.org/.
[18] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, vol. 13, no. 3, pp. 187–195, Dec. 1990.
[19] M. Kamkar, P. Krajina, and P. Fritzson, "Dynamic slicing of parallel message-passing programs," in *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, 1996. PDP '96.*, jan 1996, pp. 170 –177.

[20] J. Rilling, H. Li, and D. Goswami, "Predicate-based dynamic slicing of message passing programs," in *Second IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, pp. 133 – 142.
[21] G. Shanmuganathan, K. Zhang, E. Wong, and Y. Qi, "Analyzing message-passing programs through visual slicing," in *International Conference on Information Technology: Coding and Computing (ITCC)*, vol. 2, april 2005, pp. 341 – 346 Vol. 2.
[22] M. Strout, B. Kreaseck, and P. Hovland, "Data-flow analysis for mpi programs," in *International Conference on Parallel Processing (ICPP)*, aug. 2006, pp. 175 –184.
[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, 2005, pp. 190–200.
[24] Allinea Software Ltd, "Allinea DDT - Debugging tool for parallel computing," http://www.allinea.com/products/ddt/.
[25] GDB Steering Committee, "GDB: The GNU Project Debugger," http://www.gnu.org/software/gdb/documentation/.
[26] J. DelSignore, "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
[27] S. M. Balle, B. R. Brett, C. Chen, and D. LaFrance-Linden, "Extending a Traditional Debugger to Debug Massively Parallel Applications," *Journal of Parallel and Distributed Computing*, vol. 64, no. 5, pp. 617–628, 2004.
[28] G. Watson and N. DeBardeleben, "Developing Scientific Applications Using Eclipse," *Computing in Science & Engineering*, vol. 8, no. 4, pp. 50–61, 2006.
[29] J. Hollingsworth and B. Miller, "Parallel program performance metrics: a comparison and validation," in *Proceedings of Supercomputing '92.*, Nov 1992, pp. 4 –13.
[30] The Portland Group, "PGPROF Graphical Performance Profiler," http://www.pgroup.com/products/pgprof.htm.
[31] Q. Gao, F. Qin, and D. K. Panda, "DMTracker: Finding Bugs in Large-scale Parallel Programs by Detecting Anomaly in Data Movements," in *ACM/IEEE Supercomputing Conference (SC)*, 2007.
[32] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem Diagnosis in Large-Scale Computing Environments," in *ACM/IEEE Supercomputing Conference (SC)*. New York, NY, USA: ACM, 2006.
[33] 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/
[34] Q. Gao, W. Zhang, and F. Qin, "FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking," in *ACM/IEEE Supercomputing Conference (SC)*, 2010.
[35] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. DeRose, "Assertion based parallel debugging," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2011, pp. 63–72.
[36] D. Abramson, I. Foster, J. Michalakes, and R. Socič, "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Communications of the ACM*, vol. 39, no. 11, pp. 69–77, 1996. [Online]. Available: citeseer.ist.psu.edu/article/abramson96relative.html
[37] G. Watson and D. Abramson, "Relative Debugging for Data-Parallel Programs: A ZPL Case Study," *IEEE Concurrency*, vol. 8, no. 4, pp. 42–52, 2000.

Summary of Differences

New manuscript title: "Diagnosis of Performance Faults in Large Scale Parallel Applications via Probabilistic Progress-Dependence Inference"

A portion of the new manuscript appeared previously in the proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12) on Sep, 2012, under the title: "Probabilistic Diagnosis of Performance Faults in Large-Scale Parallel Applications".

The (previous) paper presented an algorithm (distributed) to diagnose performance faults, and a fault coverage evaluation on two HPC benchmarks.

In this paper, we present the following novel material:

- A new (semi-distributed) algorithm to diagnose performance faults. This algorithm has a higher detection precision compared to the previous (distributed) algorithm at a cost of slight higher execution time.

- A fault-coverage comparison (detection accuracy and precision) between the previous and the new algorithm—the previous paper only considered one algorithm, thus a comparison between alternate approaches was not presented.

- A performance and fault-coverage evaluation of the two algorithms on ten HPC benchmarks—the previous paper had a limited evaluation using only two benchmarks. Based on the new evaluation benchmarks, we gained insight about the type of applications the technique is suitable for.