Probabilistic Diagnosis of Performance Faults in Large-Scale Parallel Applications

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

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

§Lawrence Livermore National Laboratory, Computation Directorate, Livermore, CA 94550
{ahn1, bronis, tgamblin}@llnl.gov

ABSTRACT

Debugging large-scale parallel applications is challenging. Most existing techniques provide mechanisms for process control but little information about the causes of failures. Most debuggers also scale poorly despite continued growth in supercomputer core counts. Our novel, highly scalable tool helps developers to understand and to fix performance failures and correctness problems at scale. Our tool probabilistically infers the least progressed task in MPI programs using Markov models of execution history and dependence analysis. This analysis guides program slicing to find code that may have caused a failure. In a blind study, we demonstrate that our tool can isolate the root cause of a particularly perplexing bug encountered at scale in a molecular dynamics simulation. Further, we perform fault injections into two benchmark codes and measure the scalability of the tool. Our results show that it accurately detects the least progressed task in most cases and can perform the diagnosis in a fraction of a second with thousands of tasks.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Distributed debugging, Debugging aids

General Terms
Reliability, Performance

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 tools cannot scale to the process counts of the largest systems. More importantly, most tools provide little insight into the causes of failures. The situation is particularly challenging for performance faults (e.g., slow code regions) and correctness problems (e.g., deadlocks), which may manifest in different code or on a different process from their causes.

We present a framework that provides insight into performance faults in large-scale parallel applications. Our framework identifies the least progressed (LP) task (or tasks) probabilistically in parallel code by using a Markov Model (MM) as a lightweight, statistical summary of each task’s control-flow history. MM states represent MPI calls and computation, and edges represent state transitions (i.e., transfer of control between two code regions). This model lets us associate faults with code locations. However, in parallel applications, faults may lie on separate tasks from their root causes, so we introduce progress dependence to diagnose performance faults in parallel applications. We create 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 in parallel. Once we find the LP task, we apply program slicing [35] on the task’s state to identify code that may have caused it to fail. We implement this framework as an extension to the AutomataD (Automata-based debugging for dissimilar parallel tasks) tool [9, 23].

To ensure scalability, we use a novel, fully distributed algorithm to create the PDG. Our algorithm uses minimal per-task information, and it incurs only slight runtime overhead for the applications we tested. Our implementation of AutomataD is non-intrusive, using the MPI profiling interface to intercept communication calls, and it does not require separate daemons to trace the application as other tools do (e.g., TotalView [26], STAT [5]).

This paper makes the following contributions:
• The concept of progress dependence between tasks and its use for diagnosing performance faults;
• A scalable, distributed, probabilistic algorithm to create a PDG and to discover the LP task;
• A practical way to apply backward slicing to find the

A failure is a deviation from specification, while a fault may cause failures. A hang is a failure, while a fault is a code segment that sends an incorrect message.
We evaluate AutomaDeD by performing fault injections on AMG2006 and LAMMPS, two of the ASC Sequoia benchmarks. AutomaDeD finds a faulty task in a fraction of a second on up to 32,768 tasks. AutomaDeD accurately identifies the LP task 88% of the time, with perfect precision 86% of the time. We show that AutomaDeD can diagnose a difficult bug in a molecular dynamics code [28] that manifested only with 7,996 or more processes. AutomaDeD quickly found the fault — a sophisticated deadlock condition. The rest of the paper is organized as follows. Section 2 presents an overview of our approach and Sections 3 and 4 detail our design and implementation. Section 5 presents our case study and fault injection experiments.

2. OVERVIEW OF THE APPROACH

2.1 Progress Dependence Graph

A progress-dependence graph (PDG) represents dependencies that prevent tasks from making further execution progress at any given time. It facilitates pinpointing performance faults that cause failures such as program stalls, deadlocks and slow code regions, and in performance tuning the application (e.g., by highlighting tasks with the least progress).

A PDG starts with the observation that two or more tasks must execute an MPI collective in order for (all of) them to move forward in the execution flow. For example, MPI_Reduce is often implemented in MPI using a binomial tree for short messages [30]. 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 other (possibly delayed) tasks.

This definition formalizes progress dependence: Let the set of tasks that participate in a collective operation be X. If a task subset Y ⊆ X has reached the collective operation while another task subsets Z ⊆ X, where X = Y ∪ 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 →pd Z.

Figure 1 shows an example PDG in which task a blocks in (computation code) 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 — tasks group C, D and E block in code lines 15, 17, and 20. No progress-dependence exists between groups C and E because they are in different execution branches.

2.2 Workflow of Our Approach

Figure 2 shows the steps in AutomaDeD to diagnose performance problems. Steps 1–3 are distributed while steps 4–6 are performed in a single task.

Figure 1: Progress dependence graph example.
Figure 2: Overview of the diagnosis work flow.

(1) 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. Other work uses similar Markov-like models (in particular semi-Markov models) to find similarities between tasks to detect errors [9, 23]. Our novel framework instead uses MMs to summarize control-flow execution history. No prior work to our knowledge uses MMs to infer progress dependencies.

(2) Distributed PDG creation. When a system detects a performance fault (either *AutomaDeD* or a third-party system), *AutomaDeD* uses a distributed algorithm to create a PDG in each task. First, we use 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 $S_{others} = s_1, \ldots, s_j, \ldots, s_N$, where $s_j \neq s_{local}$. Next, each task probabilistically infers its own local PDG based on $s_{local}$ and $S_{others}$.

(3) PDG reduction. Our next distributed step reduces the PDGs from step (2) to a single PDG. The reduction operation is the union of edges in two PDGs, i.e., the union (in each step of the reduction) of progress dependencies.

(4) 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.

(5) Backward slicing. We then perform backward slicing using Dyninst [3]. This step finds code that could have led the LP task to reach its current state.

(6) 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 based on the PDG. The slice brings the programmer’s attention to code that affected the LP task, and allows them to find the fault.

3. DESIGN

3.1 Summarizing Execution History

A simple approach to save the control-flow execution history directly might build a control-flow graph (CFG) based on executed statements [20]. 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 information such as the call stack, 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 function names.

3.2 Progress Dependence Inference

In this section, we discuss how we infer progress dependence 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 on which task $t_i$ is progress-dependent. In cases when a task blocks in MPI_Wait, for example when using non-blocking operations, *AutomaDeD* uses request handlers to identify the matching progress-dependence task. We cannot infer progress dependence for MPI_ANY_SOURCE, in which case *AutomaDeD* omits this progress-dependence edge, which reflects the probabilistic nature of our approach.

*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 $x \sim y$. 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).
Figure 4 illustrates how we infer progress dependence from our MMs. 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 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 according to the observed execution, if a task is in state 3 it always must 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 so $c \xrightarrow{pd} b$.

**Resolving conflicting probability values.** When 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 $c \xleftarrow{pd} d$; our results demonstrate that this heuristic works well in practice.

The dependence between task $b$ and task $e$ is null, i.e., no progress dependence exists between them. They are in different execution branches so the forward and backward path probabilities between their states, i.e., $P(3,10)$ and $P(10,3)$, are both zero. The same holds for the dependencies between task $e$ and task $c$ or $d$.

**General progress dependence estimation.** To estimate the progress dependence between tasks $i$ and $j$, we calculate two path probabilities: (i) a forward path probability $P(i,j)$; and (ii) a backward path probability $P(j,i)$. A path probability is the likelihood of going from one state to another over all possible paths. 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 $P(i,j)$ we determine $t_j \xrightarrow{pd} t_i$ since execution goes from $i$ to $j$. 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 $P(i,j)$ we determine $t_i \xleftarrow{pd} t_j$ because execution could go from $j$ to $i$ but not from $i$ to $j$.

We cannot determine progress dependence for two cases: when both probabilities are 1.0; and when both probabilities are in the range $0 < P < 1.0$. 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 decide. For these cases, AutomaDeD marks the edges in the PDG as undefined so that the user knows that the relationship could not be determined. These cases occur 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.

**Algorithm.** Figure 5 shows our local PDG construction algorithm, which takes an MM and $statesSet$, the states of all other tasks as input. We compute the dependency between the local state and $statesSet$. We represent dependencies as integers (0: no dependence; 1: forward dependence; 2: backward dependence; 3: undefined). We save the PDG in an adjacency matrix. The function $dependence$ determines...
dependencies based on all-path probabilities (calculated in probability) and Table 1 (captured in dependenceBase-dnTable).

Table 2: Some examples of dependence unions.

<table>
<thead>
<tr>
<th>No</th>
<th>Task x → Task y</th>
<th>Union</th>
<th>Reasoning</th>
<th>OR operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( i \rightarrow j )</td>
<td>null</td>
<td>first dependence dominates</td>
<td>1+0=1</td>
</tr>
<tr>
<td>2</td>
<td>( i \rightarrow j )</td>
<td>( i \rightarrow j )</td>
<td>same dependence</td>
<td>1+1=2</td>
</tr>
<tr>
<td>3</td>
<td>( i \rightarrow j )</td>
<td>( i \rightarrow j )</td>
<td>same dependence</td>
<td>0+2=2</td>
</tr>
<tr>
<td>4</td>
<td>( i \rightarrow j )</td>
<td>( i \rightarrow j )</td>
<td>undefined</td>
<td>1+2=3</td>
</tr>
<tr>
<td>5</td>
<td>null</td>
<td>null</td>
<td>no dependence</td>
<td>0+0=0</td>
</tr>
</tbody>
</table>

The overall complexity of the algorithm is \( O(s \times (|V| + |E|)) \), where \( s \) is the number of states in \( \text{statesSet} \), and \( |V| \) and \( |E| \) are the numbers of states and edges of the MM. In practice, the MMs are sparse graphs in which \( |E| \approx |V| \), so the complexity is approximately \( O(s \times |E|) \).

**Comparison to postdominance.** Our definition of progress dependence is similar to the concept of postdominance [12] 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. In order to use 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 PDG-BASED ANALYSIS

*AutomaDeD* is implemented in C++ and uses the Boost Graph Library [2] for graph-related algorithms such as depth-first search. In this section, we focus on the implementation details that ensure scalability.

4.1 Error Detection

We assume that a performance problem has been detected, for example, because the application is not producing the expected output in a timely manner. 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 (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.

4.2 Distributed Inference of the PDG

**Helper thread.** *AutomaDeD* uses a helper thread to analyze the MM (that is built in the main thread). The core of the dependence inference, Steps 2–3 in Figure 2, is distributed while only one task (MPI rank 0 by default) performs the expensive operations in Steps 4–6. *AutomaDeD* uses MPI_THREAD_MULTIPLE to initialize MPI so that multiple threads can call MPI. On machines that do not support threads, such as BlueGene/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.

**Distributed algorithm.** The following steps provide more detail of Steps 2–3 in the workflow:

1. We first perform a reduction over the current state of all tasks to compute the \( \text{statesSet} \) of all tasks.
2. We next broadcast \( \text{statesSet} \) to all tasks.
3. Each task uses the algorithm in Figure 5 to compute its local PDG from its local state and \( \text{statesSet} \).
4. Finally, a reduction of the local PDGs calculates the union of the edges (forward or backward). Table 2 shows examples of some union results. In case 1, a dependence is present in only one task so the dependence predominates. In cases 2 and 3, the dependencies are similar so we retain it. In case 4, they conflict so the resulting dependence is undefined. We efficiently implement this operator using bitwise OR since we represent dependencies as integers.

We cannot use MPIReduce for our reduction steps (for example, 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.

4.3 Determination of LP Task

We compute the LP task (or task group) 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 [5]), it can incorrectly pinpoint the LP task, 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

**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 [35], which is valid for all possible executions. Dynamic slicing [22] 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 [21, 25, 27]. 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 suffers high costs to generate traces of each task (typically by code instrumentation) and to aggregate those traces centrally to construct the slice. Some approaches reduce the size of dynamic slices by using a global predicate rather than a variable [25, 27]. 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 [29]. For example, consider the following code fragment:

```c
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    ...
}
```

Traditional slicing on `result` in line 9 identifies statements 7, 8, and 9 as the only statements in the slice, but 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 `z`. 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 [29] 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.

**Approach.** 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 algorithm:

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 $i$. $S = S \cup s_i$ 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 diagnose a difficult bug in a molecular dynamics program that manifested only at large scale. We also perform 280 experiments to evaluate AutomaDeD in a controlled setting: 50 fault injection experiments; 160 slowdown and memory usage experiments; and 70 scalability experiments. The next sections provide the experimental settings and main results.

**5.1 Case Study**

An application scientist challenged us to locate an elusive error in ddecMD, a parallel classical molecular-dynamic code [28]. The bug manifested as a hang that emerged intermittently only when run on Blue Gene/L with 7,996 MPI tasks. Although he had already identified and fixed the error after 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. ddecMD 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 instead nominates a non-member task that belongs to a different I/O group. This mechanism led to a condition 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/reccv 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 injections

Applications. To evaluate AutomaDeD, we inject faults into two Sequoia benchmarks: AMG2006 and LAMMPS [1]. These 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.

Injections. We inject a local application hang by making a randomly selected process suspend execution for a long period to activate the timeout error detection mechanism in AutomaDeD. We use Dyninst [3] to inject the fault into the application binaries as a sleep call at the beginning of randomly selected function calls (20 user, 5 MPI). 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 resulted in errors. We use a higher proportion of user function calls because more user functions than MPI functions are used at runtime. These function calls capture a wide range of program behaviors including calls inside complex loops as well as ones at the beginning or end of the program. 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 1,000 tasks in each experiment.

Coverage results. We use three metrics to evaluate diagnosis quality: LPT detection recall, the fraction of cases in which the set of LP tasks that AutomaDeD finds includes the faulty task; isolation, the fraction of cases in which the faulty task is not detected but it is the only task in a PDG node (i.e., a singleton task); and imprecision: the percentage of the total number of tasks in the LP task set that AutomaDeD finds that are not LP tasks; we should have only one task in the set since we inject in a single task. Figure 7(a)-(b) shows two cases of correct LPT detections, which should have only the one task into which we inject the error for these experiments. A singleton task appears suspicious to a user so we consider isolation as semi-successful. Figure 7(c) shows an example of isolation — the PDG isolates faulty task \{3\} (although AutomaDeD failed to select it as the LP task).

AutomaDeD detects the LP task accurately most of the time (for AMG2006, all 20 user calls and 2 MPI calls; for LAMMPS, 19 user calls and 3 MPI calls). AutomaDeD isolates the LP task in all cases that it is not detected. AutomaDeD has low imprecision: 43 (out of 50) injections resulted in no incorrect tasks in the LP set, i.e., AutomaDeD correctly flagged the LP task only 7 times. In these 7 cases, AutomaDeD can detect the faulty task by finding the isolated task in the PDG. Only one AMG2006 case gives high imprecision (0.99) since progress dependencies are undetermined (and the PDG had only one node). Three remaining cases had low imprecision of 0.01 to 0.05. LPT detection
recall is higher for user calls than MPI calls because if a
task blocks in a computation region, the remaining tasks
are likely to block in the next communication region, which
follows the computation region in our MM with probability
one and, thus, AutomayeD is likely to detect the depen-
dence. Alternatively, if a task blocks in a communication
region, the other tasks likely block in another communi-
cation region, which is necessarily not an adjacent MM state
so AutomayeD has a lower probability of finding the LP
task. Nonetheless, AutomayeD isolates the faulty task in
all cases that it does not correctly detect the LP task.

5.3 Performance

Scalability We run AMG2006 and LAMMPS with up
to 32,768 MPI tasks on an IBM BlueGene/P and measure
the time for AutomayeD to perform the distributed part of
its analysis (i.e., Steps 2–4 in its workflow). In each code,
we inject an error close to its final execution phase in order
to have the largest possible MM (to stress AutomayeD with
the largest input graph). We used BlueGene/P’s SMP mode
in which each node has one MPI task with up to four threads.

Figure 8 shows the results of these experiments. In each
run, we measure three main steps: BUILD_PDG (Steps 2 and
3); FIND_LP_TASK (the first part of Step 4 in which the helper
thread identifies the LP task); OUTPUT (the second part of
Step 4, which post-processes the final PDG). In OUTPUT, Au-
tomaDeD eliminates duplicate edges in the PDG that may
result from the distributed merge processing of PDGs. It
also groups MPI task ranks into ranges of the form [x-y] and
adds these ranges to the corresponding PDG nodes. Figure 8
shows that FIND_LP_TASK contributes the least to the analy-
sis overhead. Intuitively, finding the LP task is simple once
we have built the PDG. BUILD_PDG is the core of the analy-

Table 3: Slowdown and proportional increase in memory
usage.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slowdown</th>
<th>Memory-usage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMMPS</td>
<td>1.39</td>
<td>6.11</td>
</tr>
<tr>
<td>AMG2006</td>
<td>1.46</td>
<td>10.36</td>
</tr>
<tr>
<td>BT</td>
<td>1.18</td>
<td>3.75</td>
</tr>
<tr>
<td>SP</td>
<td>1.67</td>
<td>5.14</td>
</tr>
<tr>
<td>CG</td>
<td>1.14</td>
<td>2.21</td>
</tr>
<tr>
<td>FT</td>
<td>1.05</td>
<td>1.01</td>
</tr>
<tr>
<td>LU</td>
<td>1.39</td>
<td>5.37</td>
</tr>
<tr>
<td>MG</td>
<td>1.04</td>
<td>1.04</td>
</tr>
</tbody>
</table>

suggests that we can trigger it at multiple execution points
with minimal impact on the application run.

Slowdown and memory usage. Table 3 shows application
slowdown and AutomayeD memory usage for AMG-
2006, LAMMPS, and six NAS Parallel benchmarks: BT, SP,
CG, FT, LU and MG [7]. We omit EP because it performs
almost no MPI communication and IS because it uses MPI
only in a few code locations. Since their MPI profiles pro-
duce small MMs, monitoring at the granularity of MPI calls
does not suit these applications. Slowdown is the ratio
of the application run time with AutomayeD to the run time
without it. Memory usage shows the proportional increase in
program heap usage when we use AutomayeD (i.e., increase
= (memory-with-tool)/(memory-without-tool)). Since Au-
tomaDeD operates as a linked 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. AutomayeD incurs little slow-
down – 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 compar-
isons) to compare states. Memory usage is moderate for
most benchmarks; the largest is AMG2006 (10.36), 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 AutomayeD creates. Benchmarks with slowdown of 1.3 (or
more) call MPI routines from different contexts more of-
ten than the others. Applications with higher slowdown
(AMG2006, LAMMPS, SP and LU) also exhibit higher Au-
tomaDeD memory usage due to the number of states.

6. RELATED WORK

The traditional debugging paradigm [6, 15, 26] of inter-
actively tracking execution of code lines and inspecting pro-
gram state does not scale to existing high-end systems. Re-
cent efforts have focused on the scalability of tools that re-
alyze this paradigm [6, 10]. Ladebug [8] and the PTP de-
bugger [34] 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 [31]
identify performance bottlenecks in parallel programs so that
time can be reduced. These tools provide the time spent
in each procedure within (critical) paths while AutomayeD
automates the analysis to find the task(s) and code regions
in which a performance fault or a correctness problem mani-
fests 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.

Automated root-cause analysis tools target general coding errors in large-scale scientific computing. Research work includes probabilistic tools [9, 13, 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 [16] or MPI coding errors [13, 14, 17, 18, 32]. These tools are complimentary to AutomaDeD as they can detect a problem and trigger AutomaDeD’s diagnosis mechanisms.

Assertion-based debugging also targets reduced manual effort. Recent work addressed scalability of parallel assertion-based debugging [11] but does not suit localization of performance faults. Differential debugging [4] 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 [33], the time and scale expenses are likely prohibitive at large scales.

The closest prior work to AutomaDeD is STAT [5], 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 [21, 25, 27]. 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 use of slicing in order to limit overhead.

7. CONCLUSIONS

Our novel debugging approach can diagnose faults in large-scale parallel applications. By compressing historic control information to limit the use of slicing in order to limit overhead.

8. ACKNOWLEDGMENTS

The authors thank David Richards of Lawrence Livermore National Laboratory for helping us to conduct the blind study on ddeMD and to validate the results.

9. REFERENCES


