Accurate Application Progress Analysis for Large-Scale Parallel Debugging

Subrata Mitra[†], Ignacio Laguna[‡], Dong H. Ahn[‡], Saurabh Bagchi[†], Martin Schulz[‡], Todd Gamblin[‡]

[†]Purdue University
{mitra4, sbagchi}@purdue.edu

[‡]Lawrence Livermore National Laboratory
{ilaguna, ahn1, schulzm, tgamblin}@llnl.gov

Abstract

Debugging large-scale parallel applications is challenging. In most HPC applications, parallel tasks progress in a coordinated fashion, and thus a fault in one task can quickly propagate to other tasks, making it difficult to debug. Finding the least-progressed tasks can significantly reduce the effort to identify the task where the fault originated. However, existing approaches for detecting them suffer low accuracy and large overheads; either they use imprecise static analysis or are unable to infer progress dependence inside loops. We present a loop-aware progress-dependence analysis tool, PRODOMETER, which determines relative progress among parallel tasks via dynamic analysis. Our fault-injection experiments suggest that its accuracy and precision are over 90% for most cases and that it scales well up to 16,384 MPI tasks. Further, our case study shows that it significantly helped diagnosing a perplexing error in MPI, which only manifested at large scale.

Keywords Parallel debugging, high-performance computing, dynamic analysis, MPI

Categories and Subject Descriptors D.2.5 [*High Performance Computing*]: Testing and Debugging; D.1.3 [*MPI*]: Parallel programming; D.4.8 [*Scientific application*]: Performance

General Terms Performance, Algorithms, Reliability, Measurement

1. Introduction

Debugging large-scale parallel applications is a daunting task. The traditional debugging paradigm [1, 6] of interactively following the execution of individual lines of source code can easily break down on the sheer volume of information that must be captured, aggregated, and analyzed at large scale. Perhaps, more importantly, even if such approaches were feasible, programmers would be simply overwhelmed by massively large numbers of threads of control and program states, which are involved in large-scale parallel applications. Instead, (semi-)automated data aggregation and reduction techniques offer much more attractive alternatives. For serial programs, several projects including Cooperative Bug Isolation (CBI) [12, 19] and DIDUCE [23] already target such techniques for bug detection and identification. However, these techniques cannot be used to debug large-scale parallel programs since they do not capture and model communication-related dependencies.

Driven by these challenges, a few recent efforts provide semi-automated techniques to debug large-scale parallel applications [18, 28]. Their key insight is that, although there is a large number of tasks in a large-scale application, the number of behavioral equivalence classes is much smaller and does not grow with the scale of the application (i.e., task counts and input data). These classes are mostly defined in terms of the control-flow graph of all involved tasks. These approaches identify the behavioral equivalence classes and isolate any task, or a small group of tasks, that deviates from their assigned behavior class.

Hangs and performance slowdowns are common, yet hard-to-diagnose problems in parallel high-performance computing (HPC) applications. Due to the tight coupling of tasks, a fault in one task can quickly spread to other tasks, causing a large number of tasks, or even the entire application, to hang or to slow down. Most large-scale HPC applications use the message-passing interface (MPI) [32] to communicate among tasks. If a faulty task hangs, tasks that communicate with the faulty task will also hang during point-to-point or collective communication that involves the faulty task. These tasks will also cause other non-faulty tasks to hang, leading the application to an entire hang.

Previous work [9] proposed the notion of *progress* of tasks as a useful model to diagnose hangs and slowdowns. Intuitively, progress is a partial order for tasks, based on how much execution a task has made in relation to other tasks. The notion of progress is useful in parallel debugging because the *least-progressed* (LP) tasks[1] often contains rich information of the root-cause (i.e., the task where the error originated). Thus, traditional debuggers can be used to inspect these LP tasks in more detail.

Several static and dynamic techniques to identify LP tasks at large scales exist. However, they largely suffer fundamental shortcomings when they are applied to HPC applications. The most relevant dynamic technique is AUTOMADED introduced by Laguna *et al.* [29]. It draws probabilistic inference about progress based on a coarse control-flow graph, captured as a Markov model, that is generated through dynamic traces. However, if two tasks are in the same loop, but in different iterations when a fault occurs, AUTOMADED may not accurately determine which task is progress-dependent on which other task. This is a fundamental drawback,

PLDI '14, June 9-11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594336

[1] Since progress is a partial order more than one task can be considered least progressed. In the following we refer to this set of tasks as LP tasks.

as most HPC applications spend most of their execution time in loops [14]. For example, in scientific applications, a common use of HPC, there is typically a large outer loop, which controls the time step of the simulation, and within it, there are many inner loops, often with deep nesting levels. Thus, AUTOMADED may fail to infer progress dependencies for a large number of fault cases in HPC applications, as we show empirically in Section 5.2.

A similar approach is the Stack Trace Analysis Tool (STAT), which was introduced by Ahn *et al.* [9]. STAT relies on static analysis and uses the concept of *temporal ordering*, which creates a partial order among tasks representing their relative progress in the logical execution space, even within a loop. To identify the LP tasks, users simply select the first task in the temporal-ordering list. The temporal-ordering feature of STAT, called STAT-TO, builds def-use chains to identify *Loop Order Variables* (LOV) and uses them to determine relative progress among tasks in different iterations of the same loop. However, there are constraints on when an LOV can be identified by STAT-TO. For example, they must be explicitly assigned in each iteration and must increase or decrease monotonically; thus, a simple `while` loop that iterates over a pointer-based linked list may not have an LOV. We show this effect empirically for a variety of applications, in Section 5.2. Even in the cases where an LOV can be identified, the overhead of static analysis needed for their identification is prohibitive for complex loops as an interactive tool.

In this paper, we present PRODOMETER [2], a novel loop-aware progress dependency analysis tool that can accurately determine progress dependence and, through this, identify the LP tasks, even in the presence of loops. It is implemented purely as a run-time tool and uses the main building blocks of AUTOMADED: It keeps track of per-task control-flow information using a Markov model. States in the model are created by intercepting MPI calls, and represent code executed within and between MPI calls. At any arbitrary point in the execution, PRODOMETER can recreate the partially ordered set of progress that each task in the application has made. It sidesteps the problem of STAT by avoiding static analysis, allowing us to keep track of the progress of a task in a highly efficient way. To achieve scalability, as in AUTOMADED, we trade off the granularity at which progress is measured—it is not done at the line-of-code granularity, but for a block with multiple lines of code.

In particular, we make the following contributions:

- A highly accurate novel run-time technique that compares the progress of parallel tasks in large-scale parallel applications in the presence of loops
- An evaluation of accuracy, precision, and performance of our technique against the two state-of-the-art approaches (i.e., STAT-TO and AUTOMADED) via fault injection in six benchmarks
- A case study that shows our proposed technique can significantly aid in localizing a bug that only manifested in a large-scale production environment

Our fault-injection experiments on six representative HPC benchmark programs shows that PRODOMETER achieves 93% accuracy and 98% precision on average, and that this is 45% more accurate, 56% more precise, and more time-efficient than existing approaches. Our overhead evaluation suggests that the instrumentation overhead of PRODOMETER slows down the target programs between a factor of 1.29 and 2.4, and its per-task memory overhead is less than 9.4MB. Further, our scalability evaluation shows that its analysis itself is also highly efficient and scales logarithmically with respect to the number of tasks, up to 16,384 MPI tasks. Finally, our case study demonstrates that PRODOMETER significantly helped diagnosing an MPI bug that affected a large-scale dislocation dynamic simulation.

[2] PRODOS is Greek for progress and METER is measure.

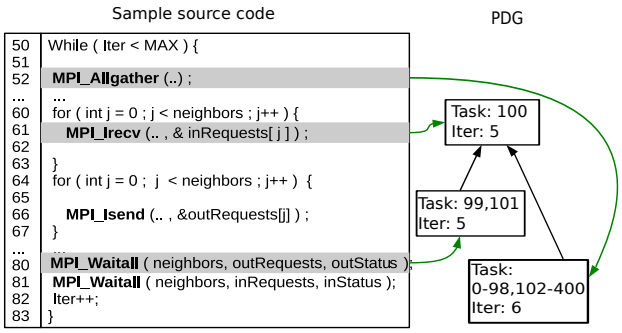

Figure 1. Iterative solver with 400 MPI tasks. Tasks are inside an outer `while` loop. Task 100 has progressed the least. Other tasks form two groups and waiting for task 100.

The rest of the paper is organized as follows. Section 2 provides a motivating example and discusses the need for accurate loop-aware analysis. Section 3 presents the overview and approach of our solution. Section 4 describes implementation details. Section 5 presents the results of our evaluation and a case study. Section 6 surveys related work. Finally, Section 8 concludes.

2. Need For Accurate Loop-aware Analysis

We detail the significance of progress dependencies as a scalable and powerful debugging idiom, as well as critical gaps in the state-of-the-art techniques that can infer them.

2.1 Progress Dependencies as a Scalable Debugging Marker

In the message-passing paradigm, parallel tasks progress in a highly coordinated fashion. They explicitly exchange messages to send or receive data relevant to their computation, and the need for matching sends and receives in point-to-point communication and collective communication calls point to the requirement of tight coordination needed for progress. For example, receivers cannot make progress until senders complete the matching send operation. This causes the progress of some tasks (e.g., receivers) to become *dependent* on other tasks (e.g., senders).

The ability to analyze such progress dependencies provides significant insight into the origin of many types of errors. Any error that disrupts the orderly progress plan of an application can reveal important clues, and thus resolving dependencies can point to the task containing the root cause.

We use Figure 1 to elaborate on this point. The code first exchanges a set of local data (e.g., ghost cells) with neighbor tasks using `MPI_Isend` and `MPI_Irecv`, non-blocking point-to-point communication calls, and then gathers computed data from *all* tasks and distributes the combined data back to *all* tasks through `MPI_Allgather`, a *collective* call. As many scientific codes model scientific phenomena over time (e.g., modeling the evolution of dislocation lines in a structure), this code iterates this computational step using the `while` loop to advance physical time steps.

Figure 1 highlights the source lines at which tasks are blocked when a single task encounters a fault (e.g., an infinite loop), showing its global impact. Specifically, task 100 triggers a fault right before executing `MPI_Irecv` and this causes its neighbor tasks 99 and 101 to form a group and to wait in `MPI_Waitall` for task 100 to complete. Meanwhile, the majority of tasks complete this loop iteration, and wait in `MPI_Allgather`, which cannot complete until the other delayed tasks can join.

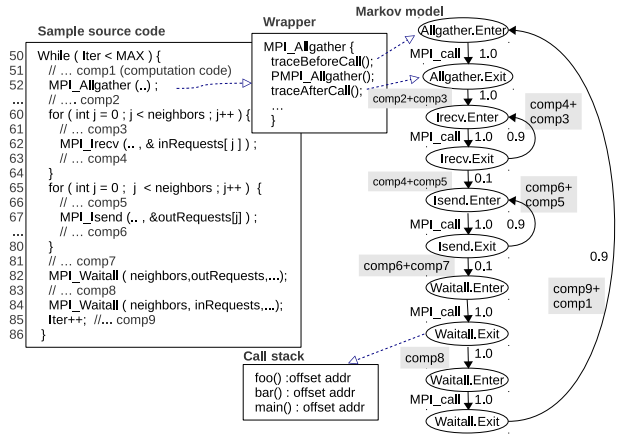

Figure 2. Markov model creation: MPI calls are intercepted and `.Enter`, `.Exit` nodes are added to the MM using the call stack *before* and *after* the actual `PMPI` calls. *Computation* and *communication*(corresponding to MPI library calls) edges alternate showing transition probabilities along them.

In general, a fault often causes tasks to form a *wait-chain* among them due to their natural progress dependencies, which eventually manifests itself as a global hang or deadlock. It is desirable to detect such conditions and to infer the dependencies automatically. Indeed, the graph on the right in Figure 1 shows that at the root of the corresponding progress dependencies lies the faulty task.

While the number of tasks in an application increases exponentially, the number of MPI calls, where tasks are stuck, is often limited. This is mainly because MPI programs are written often in a single program, multiple data (SPMD) style, which causes them to progress in an almost lock-step fashion through the same code segments, limiting the number of possible state combinations across tasks. The same holds for most multiple program, multiple data (MPMD) codes, since they are typically composed from only a small number of different programs, which are themselves SPMD. As a consequence, tasks often wait at a limited number of states, and those at the same state form a *progress-equivalence* group, which can be used as a scalable debugging marker.[3]

2.2 Markov Models as a Scalable Summary of Execution

Identifying and exploiting progress equivalence groups requires a representation of parallel program-control flow. Traditional control-flow graphs (CFGs) capture the execution flow of instructions either statically or dynamically. For MPI applications, however, control paths that capture multiple tasks and the dependencies among them, generally (except for simple programs) cannot be generated through static analysis, since the analysis of matching messages is infeasible in the general case. In contrast, CFGs created through dynamic analysis are based on the history of executed instructions and therefore implicitly capture all matched messages accurately. Nevertheless, large-scale and long-running applications can produce very large CFGs [7], presenting scalability challenges.

To overcome these challenges, Laguna *et al.* [29] used *Markov models* (MMs) as a compact, scalable summary of the dynamic execution history. They create *states* in the MM by intercepting each MPI function call, and by capturing the call stack before and after the actual call to the underlying MPI runtime (through an PMPI function call). Edges between the states (i.e., nodes) represent control-flow transitions through two types of code: (1) communication code (executed inside MPI calls), and (2) computation code (executed between two adjacent MPI calls), as depicted in Figure 2.

In addition, a *transition probability* is tracked on each edge. This probability represents the fraction of times a particular edge is traversed (out of the total number of transitions seen from that state). For example, nodes with only one outgoing edge will always transition to the next node pointed to by the outgoing edge: the transition probability is 1.0; nodes with multiple outgoing edges can have different probabilities in choosing a next node for a transition, and this depends on the previous observations. This approach provides a compact abstraction of the CFG on each nodes and can be captured even for long-running applications. Further, it can be used in subsequent steps for a scalable cross-node aggregation by forming equivalence classes of MMs.

2.3 Loops Hamper Accuracy of Dependency Analysis

Laguna *et al.* [29] used a *path probability*-based approach to resolve progress dependencies. If tasks are stuck in control-flow states S_i and S_j, they calculate probability of going from $S_i \rightarrow S_j$ as forward-path probability (P_f) and probability of going from $S_j \rightarrow S_i$ as backward-path probability(P_b). If $P_f > P_b$, then they conclude that it is highly likely that tasks at S_i eventually reach S_j (based on the execution history seen so far). Therefore tasks at S_j are more progressed than tasks at S_i. In other words, tasks at S_j are progress-dependent on tasks at S_i.

The major drawback of this approach is, however, that such inference does not work well in the presence of loops. For example, in Figure 2 the forward-path probability from the state corresponding to `MPI_Allgather` to the state corresponding to `MPI_Irecv` is 1.0 (because every task in the former eventually reaches the latter). The backward-path probability from state B to state A is either less than 1.0 (i.e., some tasks eventually exit the loop) or equal to 1.0 (a hang arises before any task exits).

In the first case, the probability-based approach will infer that tasks at `MPI_Irecv` are waiting for tasks at `MPI_Allgather`. Clearly, this is incorrect because tasks at `MPI_Allgather` have already passed `MPI_Irecv` and are waiting at the collective in the *next* iteration. In the second case, probability-based approach will not be able to infer any dependency.

Loops are very common in real-world applications, ranging from the main time-step loop to many internal computation loops. Thus, we need a highly accurate and scalable analysis technique that can resolve dependencies even in the presence of loops.

3. Approach

To address the challenges laid out above and to go beyond the shortfalls of current tools, we designed and implemented PRODOMETER, a highly accurate and scalable analysis tool that can resolve dependencies even in the presence of loops. It detects the least-progressed (LP) tasks and uses them to pinpoint the tasks where a fault is first manifested. For its analysis, it builds a dynamic progress-dependency graph (PDG), which gives insight into the progress relationships of all MPI tasks, by first creating space-efficient, per-node Markov models (MMs) capable of abstracting long-running executions, and then grouping them into progress-equivalence groups for scalability.

Figure 3 gives an overview of the PRODOMETER's workflow. Programmers link PRODOMETER, e.g., by preloading its shared library, to their MPI application (Step 1 in the figure). In Steps 2 and 3, PRODOMETER monitors the application at run-time and creates an MM for each task. The MM creation is fully distributed (i.e., no communication is involved). During the execution, PRODOMETER uses a helper thread in each task to detect hangs: when this thread detects inactivity in the main application thread (i.e., it does

[3] A more formal definition of progress dependency can be found in [29].

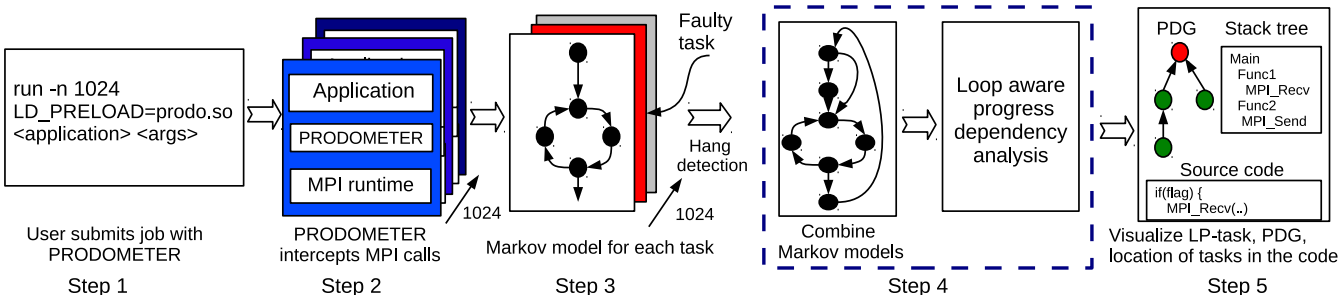

Figure 3. Workflow of PRODOMETER

not see any state transition in a configurable amount of time), it signals a fault, which triggers its analysis including the creation of the PDG (Step 4). Finally (Step 5), PRODOMETER allows users to visualize the PDG, the LP tasks, call stack trees, and annotations on the source code where different tasks are waiting.

3.1 Markov Model Creation

PRODOMETER uses MPI wrappers to intercept calls to MPI functions. Within each wrapper, it identifies the MPI call as well as the computation since the previous MPI call using the call stack observed at that point and adds each of two states to the MM, if they are not yet present. In this case, it also assigns an integer identifier to the newly created state in increasing order. Thus, this identifier represents the order in which different states are created.

While a traditional MM only keeps transition probabilities on edges [29], this model cannot capture loop iteration information. Therefore, PRODOMETER augments the MMs to keep track of inter-state *transition counts*. A transition count captures the number of times a transition between two states in MM has been observed.

Since the density of calls to MPI functions in the code is reasonably high in most applications, our technique provides appropriate granularity for localizing a problem in the code region. For the applications where MPI call density is low, a binary instrumentation technique could be used to insert additional markers. At each marker, the corresponding wrapper will insert a state in the MM increasing the granularity of diagnosis. This technique does come with some run-time overhead, but users can control it by choosing an appropriate sampling rate.

3.2 Concept of Progress

To infer progress dependencies, our algorithms treat each MM as a coarse representation of a dynamically generated control-flow graph. Thus, we assume that loop properties, such as entry and exit nodes, backedges, and domination [10] also apply to our MM analysis. In particular we assume: (1) a loop has an *entry* and an *exit* node; (2) a node x *dominates* node y, if every path of directed edges from the start node to y must go through x [16]; (3) a loop has a *backedge* (identified as an edge whose head dominates its tail), and (4) a loop with a single entry node is called *reducible* [11].

Next, we define the concepts of loop-nesting order, and relative progress, which we use later for our algorithms.

Loop-nesting order. Let L_x and L_y be two loops in an MM. Let N_x and N_y be the sets of nodes that belong to L_x and L_y, respectively. A loop-nesting order exists between L_x and L_y if $N_y \subset N_x$, i.e., all nodes in L_y also belong to L_x. Then, we call L_x has higher loop-nesting order than L_y, $L_x > L_y$. Intuitively, L_x is the outer loop in a loop nesting.

Relative progress. Let two tasks T_1 and T_2 be at node i and j in an MM. If i and j are not inside a loop then T_2 has made more progressed than T_1 if there is a *path* from i to j (i.e., it is possible that T_1 can reach T_2 by following a sequence of forward edges) but not vice versa. If i and j are inside a nesting of loops then T_2 has made more progress if loop-nesting order exists between these loops and T_2 has made more transitions along a path in the outer loops. Let $L_1, .., L_i, L_j, .., L_n$ be n nested loops, where $L_i > L_j$. Let t_{1i} denote the number of transitions made by task T_1 along loop L_i. Then, T_2 is more progressed than T_1, $T_1 \preceq T_2$, iff $t_{1i} < t_{2i}$ and $t_{1k} = t_{2k} \forall k < i$ or $t_{1k} = t_{2k} \forall k \leq n$, i.e., the lexicographical order between t_{1k} and $t_{2k} \forall k \leq n$.

Intuitively, we compute relative progress in a nesting of loops by first comparing the number of transitions made in the outermost loop, if equal, comparing the next inner loop, and so on.

Ahn et al. [9] showed that relative progress is a partial order because it is reflexive ($T_i \preceq T_i \forall i$), antisymmetric ($T_i \preceq T_j$ and $T_j \preceq T_i \Rightarrow T_i = T_j \forall i, j$) and transitive ($T_i \preceq T_j$ and $T_j \preceq T_k \Rightarrow T_i \preceq T_k \forall i, j, k$). If relative progress cannot be resolved between two tasks, we call the tasks *incomparable*. Such tasks would be executing in two separate branches in the MM. For example, relative progress order between two tasks stuck in distinct branches (e.g., *if* and *else* branches) of a conditional statement cannot be resolved, unless they are inside a loop and have completed different iterations in that loop.

3.3 Iteration Counts in Markov Model

We define the *iteration count* of a loop as the number of transitions a task has *completed*—i.e., it has traversed the *backedge* to the loop-entry point—along only that loop. Due to loop nesting, an edge belonging to a loop in an MM can also be shared by other loops. Our tool keeps track of the number of transitions along MM edges and from this, it derives the number of iterations of a loop that have been executed thus far.

Let t_i denote the number of transitions made by the program along edge e_i in the MM. If there are n loops $l_1, .., l_n$ which share this edge, and if we denote IC_k as the iteration count for loop l_k, then for the backedges, $t_i = \sum_n IC_k$, i.e. the transition count along a backedge is the summation of all the iteration counts of the loop nesting surrounding that edge. But, for an iteration in progress (not completed yet), the edges on the *forward path* of the loop will have one extra transition making the transition count greater than the summation of ICs. Thus in general we can write $t_i \geq \sum_n IC_k$.

Characteristic edge. We define the characteristic edge of a loop as the edge that is not part of any other loop. Therefore, the transition count on that edge accurately represents the iteration count of that loop. Let E_l be a set of all edges e_i which constitute loop l. Each edge might belong to more than one loop. Thus, a characteristic edge e_k of a loop x will be such that $e_k \in E_x$ and $e_k \notin E_x \cap E_y : \forall y \neq x$. In 3.4.2 we discuss how can we identify a characteristic edge for all practical loops.

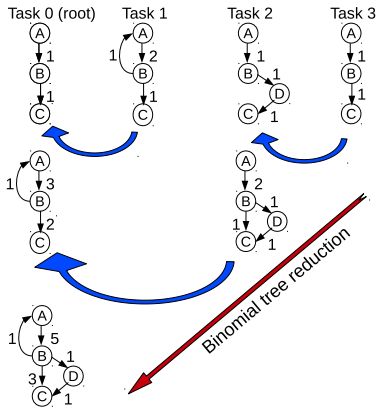

Figure 4. Aggregation of models: We aggregate MMs of individual tasks into a single (global) model by a reduction that uses a binomial-tree algorithm. Transition counts on the edges are also combined.

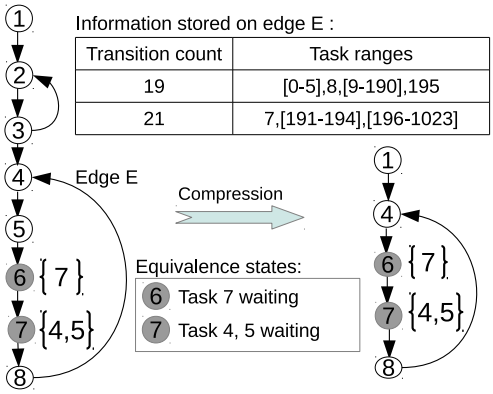

Figure 5. Tasks are grouped based on transition counts and stored in a scalable manner on each edge. Markov model is compressed to keep only useful information about control flow structure and equivalence states.

3.4 Analysis Step: PDG Creation

Definition. A progress-dependency graph (PDG) represents relative dependencies that prevent tasks from making further execution progress in case of a fault [29]. The graph shown in Figure 1 is an example of PDG which shows two task groups (99 and 101, and all the others) being dependent on task 100. Thus in a PDG, the nodes corresponding to LP tasks will be the nodes that have no further dependencies on other tasks and hence no outgoing edge.

To create such a PDG, we first need to resolve relative progress between different tasks through the following steps.

Aggregation of models. After a fault is detected, PRODOMETER begins the analysis step. PRODOMETER gathers all MMs from each task into the *root* task (rank 0) and creates an *aggregated* MM. This is done by a custom reduction operation, using individual models as input. We use an aggregated model, instead of using distributed models, because it gives us a global picture of all the states in which the MPI tasks are in and the history of control-flow paths of each task. Figure 4 shows how the aggregation algorithm creates a single MM at the root-task by combining all the states and edges of individual MMs from each task. In this figure first Task-0 and Task-2 combines MMs from Task-1 and Task-3 respectively by using the union of all the states and edges of 2 participating tasks. In the next step, Task-0 or the root task follows similar procedure to combine MM from Task-2 with its own MM to create the final aggregated MM. MM aggregation allows PRODOMETER to identify loops that could not be identified by looking at individual MMs. For example, two tasks might observe partial paths between two states in their MMs, while a global picture might reveal the existence of a loop when their per-task paths are combined.

All edges in the aggregated MM are annotated with transition probabilities and counts along with the unique identifier of the corresponding task. After the aggregation, if a state *s* has *k* outgoing edges and T_i represents the transition count for i^{th} edge, which connects to next state *r*, we calculate the transition probability from state *s* to state *r* as $T_i / \sum_k T_i$.

We keep raw transition counts corresponding to each task for subsequent analysis. To achieve scalability, instead of using a linear buffer to store transition counts for each task, we group the tasks based on transition counts—on each edge of the aggregated MM, we store unique transition counts as the *key* in a compact task list. To represent consecutive MPI ranks in a group, we use ranges of values. Thus, each entry in this representation (i.e., a table) are the tasks that have seen the same number of transitions along that edge, as shown in Figure 5. This approach makes our tool scalable; it greatly reduces the memory footprint because the number of task groups is far fewer than the number of tasks. This is because the tasks, large in number as they are, are found to be waiting in only a few places in the code. In practice, we have found the size of this table to be in the order of tens for an application with hundreds of thousands tasks.

Equivalence states. When a fault occurs, multiple tasks may be in the same state in the MM (i.e., they are executing the same code region). This behavior is due to the SPMD nature of MPI applications, and it simplifies our problem—it naturally creates *equivalence states*. Our analysis deals with equivalence classes of tasks rather than each task individually. An equivalence MM state and a set of iteration counts over all of the containing loops uniquely define a progress-equivalence group of tasks.

3.4.1 Compression

We eliminate unnecessary states from the MM before the analysis. An MM can have a large number of states because the same MPI call can be made from multiple different calling contexts. However, not all states and edges are interesting from the point of view of progress-dependency analysis. We are only interested in identifying progress dependencies between the equivalence MM states. Even though this step is not necessary for loop aware analysis, using it makes the algorithm scalable.

The compression algorithm works as follows. First, we replace all *small* loops with a single state. Small loops contain only two states, with a cycle between them, and they are created mainly by send/receive operations. Second, we merge consecutive linear chains of edges (i.e., sequence of edges with transition probability of 1.0) into a single edge. As shown in Figure 5, states between 1 and 4 were compressed after eliminating small loop between states 2 and 3. In all cases, the algorithm keeps consistent the transition probabilities of the entire MM—the sum of probabilities along all outgoing edges of a node is 1.0. But while doing the compression, PRODOMETER preserves all of the equivalence MM states and all loop structures containing them. For example, in Figure 5 it does not compress away states 6 and 7 because there are different groups of tasks which are waiting in these states.

3.4.2 Progress Dependency Analysis

The algorithm for resolving progress dependency can be divided into two cases: (1) when two equivalence states are inside some loop(s), or (2) when they do not share any common loop. For case

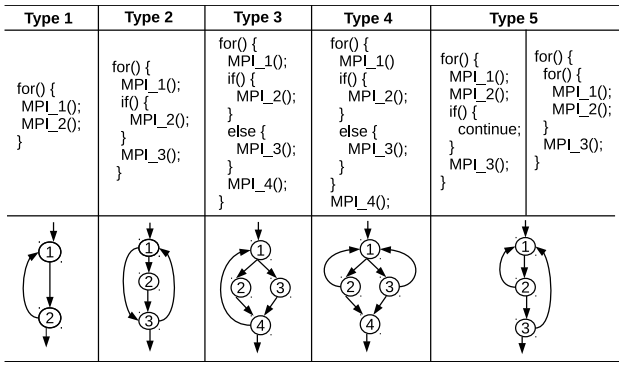

Figure 6. Categories of loops. MPI_x denotes any MPI call.

(2), PRODOMETER simply uses the algorithm in Laguna *et al.* [29]. For two equivalence states S_i and S_j, it first calculates *transitive closure*. Then for each *path* in closure it calculates forward and backward probabilities between those two states using transition probabilities present in the MM. It resolves the final dependency based on which one of these is higher. In the rest of this section, we describe how PRODOMETER resolves dependencies in case (1), our primary contribution. Algorithm 1 shows the overall procedure.

Algorithm 1 Progress dependency analysis

Input: mm: Markov model
$statesSet$: set of equivalence states where tasks waiting
Output: $matrix$: adjacency-matrix representation of PDG

```
1: procedure PDGCreation
2:     mm ← compressGraph(mm, stateSet)
3:     allLoops ← identifyAllLoops(mm)
4:     mergedLoops ← mergeLoops(allLoops)
5:     for all pair (s1, s2) in statesSet do
6:         loopSet ← getCommonLoops(s1, s2)
7:         if loopSet ≠ empty then
8:             d ← loopBasedDependency(loopSet, s1, s2)
9:         else
10:            d ← probabilityBasedDependency(s1, s2)
11:        end if
12:        matrix[s1, s2] ← d
13:    end for
14: end procedure
15: procedure getCommonLoops(s1,s2)
16:    loopSet1 ← getLoopsWithNode(s1) //loops containing s1
17:    loopSet2 ← getLoopsWithNode(s2) //loops containing s2
18:    return loopSet1 ∩ loopSet2 //return intersection:loops shared by s1 and s2
19: end procedure
```

Loop identification. PRODOMETER uses the Johnson's algorithm [26] to identify all loops in the compressed MM. This algorithm finds all the elementary circuits in a directed graph and runs in time bounded by *O((n + e)(c+1))*, where MM has *n* states, *e* edges and *c* loops. Internally, PRODOMETER uses a hash function to create an integer representation of the loop. The input to the hash function is an ordered list of the states that constitutes the loop. This integer-based representation helps PRODOMETER use faster comparisons and lookups for subsequent analysis, than if we were to use a string representation of the states.

Finding common loops. To compare two states, PRODOMETER first finds the set of loops that *contain* those states. Then, it uses our loop-aware algorithm to resolve progress dependency. If there are no common loops that contain those states, it applies case (2), as stated above. Note that HPC applications typically have multiple nested loops, which could also create a nesting of loops in the MM.

Algorithm 2 Loop aware progress dependency analysis

Input: $s1$, $s2$: Two equivalence states being compared
$loopSet$: Set of loops containing those two states
Output: Dependency relation between $s1$, $s2$ [$x \xrightarrow{d} y$ implies x depends on y]

```
1: procedure loopBasedDependency(loopSet, s1, s2)
2:     orderedLoops ← getLoopNestingOrder(loopSet) //sort loops
3:     for all loop in orderedLoops do
4:         ic1 ← getIterationCount(s1, loop)
5:         ic2 ← getIterationCount(s2, loop)
6:         if ic1 > ic2 then
7:             return s1 -d-> s2
8:         else if ic1 < ic2 then
9:             return s1 <-d- s2
10:        end if
11:    end for
12:    /* Here s1,s2 are in the same iteration for all the loops /*
13:    outerLoop ← orderedLoops.first //use only outer loop to break tie
14:    return distanceBasedDependency(outerLoop, s1, s2)
15: end procedure
16: procedure distanceBasedDependency(outerLoop,s1,s2)
17:    entry ← getLoopEntry(outerLoop)
18:    dis1 ← getDistanceFromLoopEntry(entry, s1)
19:    dis2 ← getDistanceFromLoopEntry(entry, s2)
20:    if dis1 > dis2 then
21:        return s1 -d-> s2
22:    else if ic1 < ic2 then
23:        return s1 <-d- s2
24:    end if
25: end procedure
26: procedure getIterationCount(s,loop)
27:    taskSet ← tasksWaitingAt(s) //tasks waiting at this equivalence state
28:    ic ← 0
29:    backEdgeSet ← getBackEdges(loop)
30:    for all backEdge in backEdgeSet do
31:        ic ← ic + getTransitionCount(backEdge, taskSet)
32:    end for
33:    return ic
34: end procedure
```

Loop merging. Different loops in the aggregated MM appear depending on when MPI calls are made in the source code, as Figure 6 illustrates. Our approach assumes that loops are reducible [11] (which implies that the code does not use "goto" statements, for example). Figure 6 shows the basic loop categories that PRODOMETER can handle.

In our survey of multiple HPC benchmarks and from our experience with scientific applications, we observed that most (non-goto) loop structures found in HPC applications are composed of these basic loop categories. For example, if PRODOMETER encounters a complex loop-nesting structure in the MM, it breaks it down to simpler structures, and tries to map each structure to one of these basic categories.

PRODOMETER uses purely dynamic analysis. As a result, it initially detects multiple loops in the MM corresponding to a single source-code loop. For example, in Figure 6, for Type-3, PRODOMETER initially determines one loop as 1 - 2 - 4 - 1 and a separate loop as 1 - 3 - 4 - 1. Iteration counts in these initially separated loops do not provide a complete picture and cannot be used to resolve relative progress. For example, an *if-else* statement inside a loop might appear as two separate loops in MM. To resolve relative progress between two tasks, one of which took the *if* path and the other one *else* path, we compare their iteration count in the actual source-code loop, which encloses the *if* and the *else* path.

PRODOMETER identifies different loops created from a single source-code-level loop and merges those to create a single loop which represents the original source-level loop. Assuming a reducible MM, each loop has only one *entry point* [16]. Therefore, we consider all loops with the same entry point as a single loop.

The only category of loops that creates ambiguity is Type-5. As shown in Figure 6, such MMs might be created either from a single loop through *if-continue* statements, or from two nested loops. But due to the SPMD nature of MPI applications, we do not need to distinguish between these two cases for our analysis.

Identifying characteristic edges. Due to nesting, most of the edges in an MM belong to multiple loops. Thus, a transition count on those edges corresponds to a combined total count of many different loops. This problem can be solved by solving a system of linear equations and inequalities. The unknown quantities of these equations would be the iteration counts of various loops, and the known quantities would be the transition counts of various edges. However, for practical applications, solving the system of linear inequalities is a computationally expensive procedure for a dynamic tool. PRODOMETER avoids this expensive solution by a simple observation: A loop makes a transition along the backedge(s) when it completes one iteration. Also the backedge of a loop is not shared with any other loop, after loop entry-point based merging has been done as described above. For loops with a single backedge (Types 1,2, and 3), the transition count along the backedge correctly represents the iteration count of the loop. Thus, a backedge satisfies all the properties of a characteristic edge discussed before.

An exception is the case when loops have multiple backedges (Types 4, 5). In these cases, instead of considering a single backedge, we consider the *combination* of backedges as the characteristic edge and use it to find the iteration count of the loop. Then the iteration count of the loop becomes $\sum Tb_i$ where Tb_i is the transition count on i^{th} backedge.

Lexicographic comparison. Our tool resolves relative progress between two tasks inside a complex nesting of loops by comparing iteration counts in the *lexicographic order* (i.e., in the order from outer to inner loop). This is important because there might be cases where, between two tasks inside a 2-level nesting, one task has completed more iterations on the inner loop while the other made more progress in the outer loop. In this case, we assume that the task with more iterations on the outer loop has made more overall progress. To identify outer and inner loop in an MM, PRODOMETER considers the loop whose entry state *dominates* the other. This can be simply checked using state identifiers assigned to each state. Entry-point of the outer-loop will always be created before the inner loop and therefore will have a smaller identifier.

Distance-based comparison. In some situations, two equivalence MM states may have the *same* iteration count for all nesting levels of loops. Then, PRODOMETER uses a *hop-count distance* from the loop entry-point as the metric for progress, i.e., the number of edges traversed between the entry-point of the loop and the current state. A state that has a higher value of the hop-count distance is more progressed than one with a lower value. Algorithm 2 formally describes the loop-aware analysis procedure.

Finally, a PDG is created from these pairwise dependencies between equivalence MM states. In a PDG, a directed edge goes from a *more-progressed* state to a *less-progressed* state showing their relative dependencies. Note that a state can contain multiple tasks, all of which are currently waiting in that state. A PDG is a graphical representation of the partial order.

4. Implementation

PRODOMETER is implemented in C++ as an extension to AUTOMADED's framework [29]. We implemented and tested it on x86/Linux and IBM Blue Gene/Q architectures, although the design is portable to any MPI-based parallel platform. The source code for PRODOMETER is available at [4] as part of the AutomaDeD project. In this section we discuss implementation-related aspects, in particular how we aggregate MMs in a scalable manner, how we detect faults, and how users can easily visualize the LP tasks.

4.1 Scalable Reduction of MMs

We implement a scalable binomial-tree-based algorithm, which merges MMs from individual tasks in $O(log(p))$ time, where p is the number of tasks. We cannot use `MPI_Reduce` to combine MMs because tasks can contribute states of different sizes. Since we keep an integer representation of each state, we can easily map states from different tasks into a state in the aggregated MM with efficient integer-based comparison. The merged MM contains a union of all states and edges, and thus can handle the case where individual MMs differ from one another. Figure 4 depicts this logarithmic reduction technique using an example of 4 tasks.

4.2 Fault Detection

Fault detection is orthogonal to our root cause detection in PRODOMETER. It can be combined with any technique available by the target platform or can even be done in cooperation with the application. By default, we include a platform and application-independent heuristic based on a timeout mechanism. For this we use a *helper thread* per task that monitors the application to determine if a hang has occurred. The thread *caches* a sequence of the last *N* states that the application has seen. Each time it sees a state, it checks if the state is present in the cache. If it is not, it resets a timer and inserts that state into the cache. As a result, if it does not encounter a new state for a long time and only repeatedly cycles through the states in the cache, then the timer expires and it signals a fault.

Our default technique can detect hangs arising from deadlocks, livelocks, and slow code regions. PRODOMETER can infer a reasonable timeout threshold from the mean and standard deviation of previous state transitions in the MM. Users can also provide a timeout period to account for special application characteristics. We have found in practice that a period of 60 seconds is good enough to detect a fault in most of the applications. Cache size *N* is a configurable parameter and depends on application characteristics. A low value of *N* decreases the coverage of the fault detection whereas a large value might trigger false alarm for large loops. We found a value of 10 works reasonably well for real applications.

4.3 Determination of LP Tasks

PRODOMETER computes the LP tasks from the PDG, by identifying the nodes that do not have any progress dependency, i.e., nodes with no outgoing edges in the PDG. If it finds more than one such node, PRODOMETER uses point-to-point message send information to reduce this list. For example, if it currently has both nodes i and j in the set of LP tasks and the MPI trace contains a point-to-point message from i to j, but not vice-versa, it discards node j from the LP task set due to the observation that node j expects a message from node i.

4.4 Visualization

After the analysis, PRODOMETER opens a graphical interface to visualize the PDG as a graph. The LP tasks are highlighted with different colors. It also marks if the bug was identified in a communication node or computation node with different shapes of nodes. Users can interact with the graph by selecting one or multiple nodes, which will show a *parallel stack tree* of call-graphs and highlight corresponding lines in a source code viewer.

5. Evaluation

In this section, we show accuracy and precision of PRODOMETER using controlled experiments, followed by a real world case study.

5.1 Setup of Controlled Experiments

To evaluate the effectiveness of PRODOMETER, we set up controlled experiments in which we dynamically inject faults into applications, and measure its precision and accuracy in identifying the task that was injected. We compare our results to two existing state-of-the-art techniques, STAT-TO and AUTOMADED.

We implement the fault injection using the binary instrumentation library PIN [3] and use it to randomly inject an infinite loop as the fault at runtime. To cover a wide range of HPC application patterns, we choose three applications (AMG, LAMMPS and IRS) from the Sequoia procurement benchmark suite [5], a widely studied proxy application (LULESH [2]), and two programs from the NAS parallel benchmark (BT and SP), totaling six programs.

As commonly found in real-world HPC applications, most of these benchmark programs have two distinct simulation phases: a setup and a solver phase. During the setup phase, they generate their basic data structures, e.g., a mesh, and distribute the input data across MPI tasks. Once done, they move to the solver phase where the tasks start to iterate through a time-step or solver loop and solve the given problem. While production applications spend most of their simulation time in their solver phase [15], these benchmark programs can spend a relatively large portion of time in the setup phase, due to relatively small input data set sizes as well as artificially reduced iteration counts, which makes them more suitable for experimentation and procurement testing while not changing the computational characteristics in each phase. To compensate for this bias, we inject faults only into the solver phase.

We first run each of these programs under PIN and profile all functions invoked in its solver phase. We filter out function calls from within well-known libraries, like `libc`, MPI and math libraries, to capture the fact that faults are more likely to be in the application than in these well-known and widely tested libraries.

We then randomly select various parameters to make our fault injection campaign statistically fair. Of all unique functions found in the profile, we randomly select 50 functions, and then pick one invocation of one of these functions for injection—this ensures we inject a fault into a random iteration of a loop. Similarly, we select one task out of all of the MPI tasks as the target for this injection.

Finally, we run these programs at different scales to observe any scale-dependent behavior of our technique. We use 128, 256 and 512 tasks for the cases, where the programs do not have restrictions on the task count to use; for some other benchmarks such as LULESH, IRS and BT, which have specific restrictions, we use the closest integers to these counts.

5.2 Accuracy and Precision

We use two metrics to summarize the findings of our controlled experiments and to quantify the quality of root cause analysis: *Accuracy* and *Precision*. *Accuracy* is the fraction of cases that a tool correctly identifies the Least-Progressed (LP) tasks. *Precision* is the fraction of the identified LP tasks that are actually where the fault was injected. Since we inject a fault only into a single task, ideally PRODOMETER should detect only one task as the LP task.

In the first study we compare PRODOMETER to AUTOMADED. As mentioned above, AUTOMADED uses a similar approach in gathering runtime statistics using MMs, but is not capable of dependencies across loop boundaries.

Table 1 summarizes the accuracy results for PRODOMETER and AUTOMADED. PRODOMETER achieves over 93% accuracy on average, across all tested programs and scales, and its accuracy is not affected by scale. Further, the data shows that PRODOMETER's accuracy is significantly higher than that of AUTOMADED (64%). This is mainly because faults are injected into the solver phases which typically contain many complex loop-based control flows. Nevertheless, the accuracy of AUTOMADED, which does not have a special logic to infer progress inside a loop, is not close to zero, even on those programs with time-step loops governing the entire solver phase. This can be caused by faults that prohibit the completion of even a single iteration of the time-step loop. Thus, from the perspective of the Markov model, the loop was never entered, and AUTOMADED could infer progress of this region as if there was no loop. For BT, accuracy of PRODOMETER is relatively low, which is caused by the use of *goto* statements inside loops. Our current loop-detection algorithm is based on finding "natural loops", i.e., loops with a single head node and a backedge in the CFG. The *goto* statement violates this assumption, and we leave support for such cases to our future work.

	128 tasks		256 tasks		512 tasks	
Benchmarks	PR	AU	PR	AU	PR	AU
LAMMPS	1.00	0.54	1.00	0.48	1.00	0.58
AMG	0.92	0.56	0.94	0.46	0.88	0.67
IRS	1.00	0.50	1.00	0.76	1.00	0.78
LULESH	0.90	0.60	0.90	0.60	0.92	0.56
BT	0.82	0.52	0.84	0.66	0.84	0.68
SP	0.94	0.80	0.92	0.82	0.92	0.82

Table 1. Accuracy: PRODOMETER (PR) vs. AUTOMADED (AU)

	128 tasks		256 tasks		512 tasks	
Benchmarks	PR	AU	PR	AU	PR	AU
LAMMPS	0.98	0.75	0.99	0.68	0.98	0.47
AMG	1.00	0.89	1.00	0.73	0.99	0.71
IRS	0.96	0.54	0.98	0.67	0.97	0.75
LULESH	0.97	0.46	0.98	0.25	0.94	0.28
BT	0.98	0.67	1.00	0.63	1.00	0.42
SP	1.00	0.87	0.98	0.84	1.00	0.74

Table 2. Precision: PRODOMETER (PR) vs. AUTOMADED (AU)

Table 2 shows the summary of the precision results. PRODOMETER detects LP tasks with very high precision (above 98% on average), which means that in most cases, PRODOMETER will point the developer to a single task, which she can focus on for purposes of debugging, using standard single process debuggers.

However, we believe that there are fundamental limits to the precision of any tool that determines progress dependence. This is because the concept of progress dependency is itself a partial order, and thus there exist cases where states simply cannot be ordered. Notably, one cannot resolve the ordering of two tasks that are executing in distinct branches of a conditional statement, in the same iteration count. In this case, PRODOMETER may identify both tasks as LP, which affects precision. PRODOMETER's mechanism for determining forward- and backward-paths is probabilistic, and if the prior observations are not representative enough or large enough, these introduce errors in the analysis.

Second, we compare the accuracy of PRODOMETER with STAT-TO, which is, to our knowledge, the only existing tool that is capable of finding loop dependencies. This is done in STAT-TO by detecting *Loop Order Variables* (LOVs) (which govern loops) via static analysis. Since STAT-TO requires a manual intervention and guidance, we compare the two tools by applying STAT-TO manually to some of the experiments for which PRODOMETER has succeeded. We first randomly select five cases from each of three benchmark programs (AMG, LAMMPS and LULESH), which PRODOMETER analyzed correctly using the new technique (i.e., cases with loops). Manual inspection reveals that the selected cases involve 1–3 structured loops (e.g., `while`) for each benchmark and 1–3 program points (i.e., a line in a source file) for each loop. In addition, we find that only a single program point is within a *single* loop, while all others are inside triple-nested loops.

Codes	Loops	Points	LOV	Secs
AMG	PCG solver	498,	`i`	9.0
	Coarsening	595, 609, 1183	`level`	294.1
	Coarsening	1221, 1292	`level`	295.6
	V-Cycle	237	`cycle_count`	10.5
	Main cycle	263, 335	Not found	14.6
LAMMPS	Input	187	Not found	4.6
	Verlet	206, 264	`i`	3.6
	Verlet	206, 253	`i`	3.6
LULESH	Time step	2775, 2776	Not found	13.17

Table 3. STAT-TO accuracy and performance

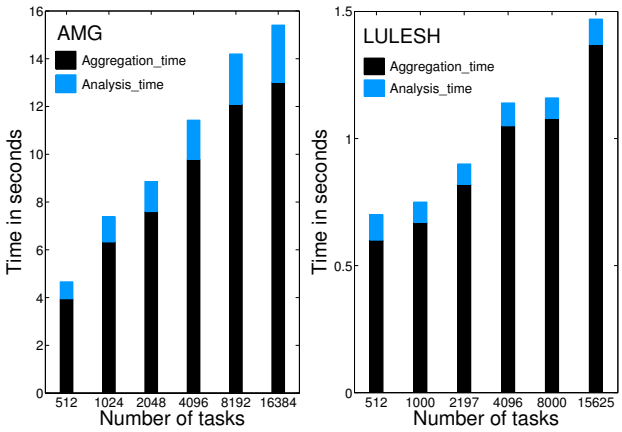

Figure 7. Scalability of PRODOMETER progress-analysis time

Benchmarks	Slowdown	Memory overhead (in MB)
AMG	2.4	9.4
LAMMPS	1.3	3.67
IRS	1.29	4.7
LULESH	1.44	2.2

Table 4. Slowdown and memory overhead for model creation

Then, we manually apply STAT-TO's Loop Order Variables (LOV) analysis to those program points that are contained in a loop. This represents the static analysis step in STAT-TO, which is most essential to resolve temporal order of the program points within loops. Further, STAT-TO requires a set of program points to be analyzed together for ordering, and thus we apply this analysis to sets of program points involved in each case. This amounts to nine distinct sets of LOV analysis runs summarized in Table 3.

In terms of accuracy, this static analysis fully retrieves a LOV in six out of the nine cases—66% retrieval rate. It completely fails to identify LOVs for two cases: one in LAMMPS and one in in LULESH. But for one case—i.e., `Main-cycle Loop`—where the program points are included in triple-nested loops, it partially fails to identify LOVs: it fails to retrieve a LOV for the outermost loop while successfully identifying LOVs for both of the inner loops. To complete temporal ordering, however, STAT-TO must fully resolve all of the loops so we log this case as *Not found.*

In terms of performance, for all but one case, LOV analysis finishes its analysis in under 15 seconds, which would be acceptable to support even an interactive tool like a parallel debugger. However, for AMG's coarsening loop, it jumps to 295.6 seconds, a factor 20 larger overhead than other loops. We find that this is due in large part to the high complexity of this loop, which triggers a longer static analysis. The def-use table used in STAT-TO exhibits over one hundred variables defined outside the loop while being used inside the loop, and over thousand references to these variables from within the loop body. Given the complexity of a def-use chain analysis algorithm, $O(N^2 * V)$ where (N) is the number of definitions and uses and (V) is the number of variables, this case has the computation complexity of $O(10^8)$. This suggests that a static analysis technique can become unwieldy, as the complexity of target loops becomes higher.

5.3 Performance and Scalability

Our second set of experiments targets the run-time overhead of PRODOMETER, in terms of execution time and memory use with the target programs. We define *slowdown* as the ratio of times it takes for the program to complete with and without PRODOMETER. Memory overhead is the memory consumed by the tool. Since different tasks can have different memory usage, we use the average number across all the tasks. Table 4 summarizes the results measured with 512 tasks for the four largest codes: the three Sequoia benchmarks and LULESH.

PRODOMETER is a dynamic analysis tool, and its interception of each MPI call followed by a system call to capture a call path is the primary reason for the increased run-time overhead. Nevertheless, the overhead is still reasonable, in particular for a debugging tool, and—most importantly—small enough to still enable the execution of full scale applications with realistic input sets. Memory overhead is a function of the number of unique states and edges in the Markov model. PRODOMETER stores call-path information in each state, and keeps track of the number of transitions on each edge.

In this experiment we statically linked the library and used return addresses from GNU *backtrace* utility to represent a call-path. Statically linking ensures that object code is loaded at the same addresses on all tasks. With dynamic linking, and with static linking on operating systems that use security features like address space randomization (typically not used on HPC systems, but default for many desktop OSs), libraries' load addresses can vary from task to task. To properly identify equivalence states across all tasks, we normalize the addresses in call-paths by representing them as a tuple (M, O) where *M* is the name of the module or library containing the address and *O* is the offset within that library. With the use of this normalization feature, we have observed slowdowns of up to 4.5x, when libraries are dynamically linked (and on systems for which this normalization feature is needed). The highest slowdown occurs for AMG. We plan to address this problem in our future work by implementing more efficient normalization and by using a faster stack tracing tool (such as libunwind).

Scalability The final set of controlled experiments evaluates the scalability of PRODOMETER's progress analysis itself by measuring model-aggregation and dependency-analysis performance. We perform this test with AMG and LULESH with up to 16,384 tasks on an IBM Blue Gene/Q (BGQ) machine. Each BGQ compute node consists of 16 PowerPC cores with 16GB of RAM, connected through a custom 5-D torus network. For our scalability test, we inject a fault close to the final execution phase of the programs so that an analysis must handle the largest Markov model. Our objective is to evaluate how efficiently PRODOMETER aggregates large Markov models from a large number of tasks and analyzes this aggregated model to determine progress dependence.

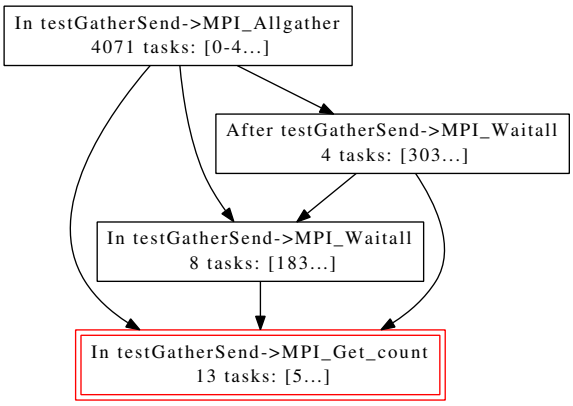

Figure 8. PRODOMETER on Dislocation Dynamics Reproducer

Figure 7 summarizes the scalability results. *Aggregation_time* denotes the time it takes for PRODOMETER to aggregate Markov models gathered from all tasks using a binomial tree-based reduction technique. *Analysis_time* denotes the time taken to identify relative progress, to build a progress dependence graph and to identify LP task(s). *Aggregation_time* increases with scale for both benchmark programs, and the trend is logarithmic with the R^2 value of a logarithmic fit (with $alog_2x + b$) is 0.98 for AMG and 0.96 for LULESH. As for *Analysis_time*, that of AMG increases with scale while LULESH stays almost constant. In the case of LULESH, the complexity of the application does not change with scale and thus the number of states remains constant, while in the case of AMG the algorithmic complexity grows with scale (e.g., the number of levels in the multi-grid method increases with scale) and thus PRODOMETER must handle larger numbers of states at larger scales. Nevertheless, the worse-case overall time is less than 16 seconds, which is quite tolerable as an automated tool for debugging.

5.4 Case Study: Using PRODOMETER on a Real MPI Bug

A dislocation simulation code recently encountered intermittent hangs during production runs on our IBM BG/Q machine soon after our computing facility had rolled out a new driver, which included a new version of the MPI library. The cause of the problem was unknown. We observed this issue more frequently at larger scales. For instance, it almost always showed up for runs with 32,768 tasks. The scientist who was developing this code reported the issue to a system analyst. He then extracted its control flow and communication patterns, and put together a highly deterministic reproducer at a reduced scale.

To help diagnose this issue further, we applied PRODOMETER to this reproducer. Figure 8 shows the global state PRODOMETER captured when the reproducer code was hung at 4,096 MPI tasks. The tool immediately helped us understand the global hang state without overwhelming us, as it expresses the state in a form of progress-equivalence classes (i.e., nodes). While this program was run at 4,096 tasks, our tool showed the state with only four progress-equivalence classes with dependencies (i.e., edges) among them.

Clearly, this diagnosis shows the reason for the global hang: the majority of the tasks (4,071 tasks) were not making progress because of their dependencies on a small number of tasks (25 tasks). Further, PRODOMETER identified the group that are still in an MPI communication routine called `MPI_Get_count` as the least-progressed group. With this information, it was likely that the root cause of this hang would be in the vicinity of the code that these less-progressed groups were executing.

Given that this reproducer was not hung under the older MPI drivers, and that it was written simply and in a way to avoid elusive non-deterministic concurrency or memory bugs, we immediately suspected a bug in the underlying communication layer itself.

Indeed, using the same reproducer, the IBM software team quickly discovered a software bug in the communication layer of their new driver whereby a new collective communication optimization was too aggressive and was causing other concurrent communications to starve. As shown in Figure 8, large numbers of tasks reached and started `MPI_Allgather` first, and this large-scale collective communication significantly starved the communication subsystem of those tasks that were still performing logically earlier point-to-point communications. In fact, the reproducer actually injects a random delay prior to certain point-to-point calls into a small number of tasks to induce this condition more frequently.

Manual analysis would have been far more confusing, since `MPI_Allgather` appears earlier in the source listing. While it is obvious that this collective call is included in the main time-step loop in the reproducer code, it is far less obvious in the real case with the full dislocation dynamics code where this collective call is buried in a function being called by an upper-routine loop.

6. Related work

Debugging and root cause analysis Debugging is one of the most crucial and time consuming processes in software development cycle. Traditional breakpoint based debugging with GDB or "print debugging" is particularly not suitable for large scale parallel applications. Parallel debuggers such as Totalview [6] and DDT [1] control multiple processes and aggregate distributed states. However, identifying the faulty process or finding the matching code location still requires interactive manual effort. Recent research on semi-automated statistical debugging has produced tools for sequential codes [13, 19, 25, 35] that, in the presence of sufficient historical data, can diagnose the root cause of a bug. Other techniques include use of boolean SAT and MAX-SAT [27, 34] for detecting program errors. Even though these techniques are quite promising, it is difficult to immediately apply those to debug parallel applications at large scale. Some formal verification based tools [33] and assertion based techniques [20] can overcome scalability challenges and adapt to parallel applications. However, these tools are mainly suited for debugging accuracy problems and are complementary to our approach. Laguna *et al.* [28] and Mirgorodskiy *et al.* [30] both monitor applications timing behavior and identify processes that exhibit unusual behaviors. DMTracker [22] uses statistical technique to find bugs in MPI applications by identifying anomalous data movements. There are other techniques [21, 24] that target general MPI coding errors and deadlock detection. These tools are also complimentary to our approach and can be used to detect a problem and trigger PRODOMETER for further analysis. The closest prior work that follows a similar aspect of relative progress as PRODOMETER are AUTOMADED [29] and the temporal ordering extension of STAT [9]. While AUTOMADED suffers from significant drawback of not being able to handle the common case — analysis in the presence of loops, STAT's static analysis based algorithm suffers from extensive static analysis times while building def-use chain and fails in the absence of loop-order-variables. Our loop-aware dynamic technique addresses both of these issues.

Loop analysis Loop analysis is an established field in compiler technology. There are many well accepted algorithms for identifying natural loops in the program and used in compilers for loop-unrolling [17], tiling [8], resolving dependency between different variables [16]. These techniques are mainly based on static analysis of the program and the goal is to improve parallelism and cache behavior. Other studies [31] use loop characterization at the hardware

level to improve branch prediction and parallelism. Our goal for dynamic analysis of loops is fundamentally different. We perform our loop-analysis on Markov models in-order to extract information about iteration count and loop nesting. We then perform lexicographical order based comparison to resolve progress between different groups of tasks.

7. Conclusion

Our novel loop aware progress dependency analysis technique can diagnose faults in large scale HPC applications with high accuracy. These are faults, like hangs and performance slowdowns, that are a dominant class of software problems encountered in HPC applications. This fully dynamic technique is easy to use and does not require modifications to the application. Its ability to handle complex loops and its approach based on runtime analysis makes it more accurate and precise in debugging complex applications, compared to existing state-of-the-art techniques [9, 29]. Further, we achieve high scalability by using Markov models to summarize the application's dynamic control-flow as well as deploying a binomial reduction of the models across tasks. Our fault injection study on 4 major applications and 2 NAS parallel benchmarks show that the least-progressed task identified through this technique can be effectively used to identify the root-cause, i.e., the faulty task and corresponding code region. On average PRODOMETER achieved over 93% accuracy and 98% precision. The case study presented in this paper shows how this technique was able to diagnose an unknown non-deterministic bug, reproducible only at large scale, in a full scale dislocation dynamics simulation code.

Acknowledgments

We thank the anonymous reviewers for their invaluable feedback. We thank Gregory L. Lee from LLNL for helping us with STAT. This work was performed partly under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-646258).

References

[1] DDT - Debugging tool for parallel computing. `http://www.allinea.com/products/ddt/`.

[2] Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). `https://codesign.llnl.gov/lulesh.php`.

[3] Pin - A Dynamic Binary Instrumentation Tool. `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[4] PRODOMETER source code. `https://computation-rnd.llnl.gov/automaded/`.

[5] Sequoia Benchmarks. `https://asc.llnl.gov/sequoia/benchmarks/`.

[6] TotalView Debugger. `http://www.roguewave.com/products/totalview.aspx`.

[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[8] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *SC*, 2000.

[9] 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*, 2009.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

[11] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 1976.

[12] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *ECML*, 2007.

[13] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[14] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[15] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, pages 345–420, Dec. 1994.

[16] U. Banerjee. Loop transformations for restructuring compilers: The foundations. 1993.

[17] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS*, 1997.

[18] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. d. Supinski, D. H. Ahn, and M. Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *DSN*, 2010.

[19] T. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[20] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. DeRose. Assertion based parallel debugging. In *CCGRID*, 2011.

[21] C. Falzone, A. Chan, E. Lusk, and W. Gropp. Collective error detection for mpi collective operations. *Recent Advances in Parallel Virtual Machine and Message Passing Interface Lecture Notes in Computer Science*, pages 138–147, 2005.

[22] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *SC*, 2007.

[23] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[24] W. Haque. Concurrent deadlock detection in parallel programs. *International Journal of Computers and Applications*, pages 19–25, 2006.

[25] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[26] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, pages 77–84, 1975.

[27] M. Jose and R. Majumdar. Cause clue clauses:error localization using maximum satisability. In *PLDI*, 2011.

[28] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree. Large scale debugging of parallel tasks with automaded. In *SC*, 2011.

[29] I. Laguna, D. H. Ahn, B. R. d. Supinski, S. Bagchi, and T. Gamblin. Probabilistic diagnosis of performance faults in large-scale parallel applications. In *PACT*, 2012.

[30] A. Mirgorodskiy, N. Maruyama, and B. Miller. Problem diagnosis in large-scale computing environments. In *SC*, 2006.

[31] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *CF*, 2007.

[32] The MPI Forum. MPI: A Message Passing Interface. `https://http://www.mpi-forum.org/`, 1993.

[33] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *SC*, 2009.

[34] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, 2005.

[35] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML*, 2006.