Scalable Parallel Debugging via Loop-Aware Progress Dependence Analysis *

Subrata Mitra[1], Ignacio Laguna[2], Dong H. Ahn[2]
Todd Gamblin[2], Martin Schulz[2], Saurabh Bagchi[1]
[1]Purdue University [2]Lawrence Livermore National Laboratory
{mitra4, sbagchi}@purdue.edu, {ilaguna, ahn1, tgamblin, schulzm}@llnl.gov

ABSTRACT

Debugging large-scale parallel applications is challenging, as this often requires extensive manual intervention to isolate the origin of errors. For many bugs in scientific applications, where parallel tasks progress forward in a coordinated fashion, finding tasks that progressed the least can significantly reduce the time to isolate error root causes. We present a novel run-time technique, the loop-aware progress-dependence analysis, that improves the accuracy of identifying the least-progressed (LP) task(s). Our technique extends an existing analysis technique (AutomaDeD) to detect LP task(s) even when the error arises within complex loop structures. Our preliminary evaluation shows that it accurately finds LP task(s) on several hangs, where the previous technique failed.

1. RELATIVE PROGRESS ANALYSIS

Debugging large scale parallel applications is difficult. On elusive errors, such as hangs, deadlocks, and slow codes, the traditional paradigm of interactively following the execution of individual source lines breaks at large scale. Due to a large search space, programmers can no longer rely on their intuition to find the parallel tasks that have fallen behind—an essential information to find the root cause.

The (semi-)automatic debugging paradigm [4, 2] provides an attractive alternative: It can streamline the detection and root-cause analysis of subtle errors with minimal manual effort. Tools that embody this paradigm must automatically and scalably detect an error at run-time, analyze relationships among the tasks, and present to users the potential error origins without overwhelming them.

The ability to relate parallel tasks on their relative progress [1, 3] has been effective on localizing the exact code region where the error originated. Disruption of the orderly temporal plan of an application, that progresses in a coordinated fashion, can often reveal important clues. However, it is non-trivial to devise a scalable (yet highly accurate) run-time analysis that automatically extracts the progress dependence of tasks at arbitrary points in execution.

2. BACKGROUND: AUTOMADED

The progress dependence analysis of AutomaDeD [3] provides insight into performance faults in parallel applications. It probabilistically identifies the LP task(s) by comparing the Markov model (MM)—a compact history of control-flow—of each MPI task. States in an MM represent MPI calls and computation, and edges represent transfer of control between two states. In parallel applications, a fault might lie in a different task from its root cause. Thus, AutomaDeD creates a progress-dependency graph (PDG) that captures the wait chain of non-faulty tasks that depend on the faulty task to progress. This groups a large number of processes into a smaller number of equivalence classes, which relate to one another through progress dependence.

AutomaDeD calculates progress dependence based on transition probabilities between states. Each task first creates locally a MM, where each edge has a transition probability according to the transition history. Using *forward* and *backward* path probabilities, AutomaDeD infers a dependency between two states (at which tasks are executing). If the forward path probability from state A to state B is greater than the backward path probability from state B to state A, then A was expected to reach B, and hence B is waiting (depending) on A. However, it cannot infer such dependencies where both probabilities are zero (i.e., no path exists between states) or where both are a fraction.

3. ACCURACY PROBLEMS ON LOOPS

While simple and scalable, the current AutomaDeD approach fails to analyze three important patterns involving loops: (1) infinite loops (i.e., loops with no exit edges where both forward and backward probabilities in the MM are 1.0), (2) tasks are stalled during different iterations inside a loop (since the tool can not resolve progress based on loop iteration, it cannot accurately infer dependencies), and (3) two states in the local MMs are part of a loop, which becomes evident only when MMs are combined.

4. LOOP-AWARE ANALYSIS

We extend AutomaDeD to address its current limitations by adding a loop analysis support to MMs. AutomaDeD with our loop-aware progress-dependence analysis is denoted as AutomaDeD-LA. Our approach consists of three major steps: (1) augmenting local MMs with loop information, (2) scalable merging of local MMs to a global MM, and (3) an algorithm for loop-aware detection of progress dependencies.

4.1 Augmenting Markov models

The number of transitions between two adjacent states in an MM will be more than one only if the connecting edge is part of a loop. Therefore, we can indirectly get information

*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-ABS-641596).

Table 1: Evaluation Results

Bug Type	Application Type	AutomaDeD output	AutomaDeD-LA output
Deadlock	Producer-Consumer	Incorrect PDG (due to infinite loop)	Correct PDG
Hang	Poisson Solver	Could not create PDG (due to infinite loop)	Correct PDG
100 randomly injected hangs	AMG	82% accuracy	100% accuracy

about loop iterations from transition counters. Thus, we augment each edge of local MMs with a transition count.

4.2 Scalable reduction of Markov models

We first create a global MM by merging local MMs from individual tasks in a scalable manner. To perform an iteration-based analysis on the merged MM, we want to preserve transition numbers on a particular edge for all the tasks that contributed to that edge. For scalability, we group tasks based on same transition counts on each edge and maintain a compact tasks list (i.e., rank ranges) per unique count.

Algorithm 1 PDG creation

Input: mm: Markov model

$statesSet$: set of current states of all tasks

Output: $matrix$: adjacency-matrix representation of PDG

```
1: procedure PDGCreation
2:     mm ← compressGraph(mm, stateSet)
3:     for all pair (s1, s2) in statesSet do
4:         loops ← commonLoops(s1, s2)
5:         if loops is empty then
6:             d ← probabilisticDependency(s1, s2)
7:         else
8:             d ← loopDependency(loops, s1, s2, mm)
9:         end if
10:        matrix[s1, s2] ← d
11:    end for
12: end procedure
```

4.3 Loop-aware progress analysis

An edge belonging to a loop in an MM can also be contained in other loops through nesting. It is generally difficult to break down the total transition count on that edge into the individual iteration counts for the loops. Thus, we first discover the *characteristic edge* of each loop to get the accurate iteration count for this loop. 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 purely represents the iteration count of that loop. It turns out the *backedge* of a loop satisfies the criteria of a *characteristic edge*. Our loop analysis technique builds upon various concepts borrowed from compilers domain.

Algorithm 1 shows the overall loop-aware progress-dependency resolution scheme. We first compress chains of singleton states (i.e., states with one incoming and outgoing edges), which do not affect our analysis. Next, to find dependency between two task groups waiting at two different states, we find all of the common loops that contain these states. We then resolve the progress dependency of these two states by comparing the iteration counts of the common loops in the lexicographical order. For this purpose, we use a characteristic edge to get the iteration count for each loop that contains each task group. Based on the lexicographical ordering, we determine that a task group with more iterations has further progressed and is therefore dependent on other less-progressed task groups.

Table 2: Performance Results

Benchmark	Slowdown	AutomaDeD Memory overhead
AMG	1.9	2.1
LAMMPS	2.2	1.6
LULESH	1.2	6.5

To calculate the LP task(s), we resolve pairwise dependencies between all of the task groups that are waiting at different states (i.e., different code regions). If two tasks are stuck in the same iteration of a loop, there will be a tie. To resolve dependencies in such cases, we calculate the distance of a task from the *loop entry point*. The task with a shorter distance is the LP task. The entry point of a loop is the state where the *backedge* meets the starting state. If two states do not have any commonly-containing loop, we use the original probabilistic algorithm [3].

5. PRELIMINARY RESULTS

We tested several scenarios on which AutomaDeD failed to show correct dependencies. We also injected 100 hangs inside random functions in AMG. In all cases, AutomaDeD-LA was able to build accurate PDGs. Table 1 summarizes the results. We also measured the slowdown and memory overhead of AutomaDeD-LA for 3 benchmarks: AMG, LAMMPS and LULESH. Table 2 presents the numbers.

AutomaDeD-LA incurs slightly higher analysis overhead compared to the baseline AutomaDeD because it has to maintain transition counts and requires a global reduction of MMs. But even with this, AutomaDeD-LA creates PDGs in a minute from the detection of a fault with 8,192 tasks.

6. CONCLUSION AND FUTURE WORK

We propose a loop-aware progress-dependence analysis as an accurate means to extract progress dependencies of parallel tasks without compromising performance and scalability. Our novel algorithm identified the LP tasks on some synthetic—yet commonly occurring—bugs on which the baseline analysis failed. Our fault injection study confirms the accuracy of our tool. We are in early stages of this research, and future work includes extending our technique to support more complex loops and control-flow structures and evaluating its effectiveness and performance on controlled fault-injection experiments.

7. REFERENCES

[1] D. H. Ahn, B. R. D. Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable Temporal Order Analysis for Large Scale Debugging. In *SC '09*, 2009.

[2] 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.

[3] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin. Probabilistic diagnosis of performance faults in large-scale parallel applications. PACT '12, pages 213–222, 2012.

[4] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Ahn, M. Schulz, and B. Rountree. Large scale debugging of parallel tasks with automaded. In *SC*, pages 50:1–50:10, 2011.