Accurate Application Progress Analysis for Large-Scale Parallel Debugging

Subrata Mitra, Saurabh Bagchi, Purdue University
Ignacio Laguna, Dong H. Ahn, Martin Schulz, Todd Gamblin, LLNL

PLDI 2014
June 10, Edinburgh, UK
Debugging large-scale parallel programs is hard

Applications run with millions of threads. Inspecting the state of a massive number of threads and processes overwhelms developers.

Serial debugging techniques don’t work. They do no capture communication dependencies between multiple processes.

Some bugs only manifest at large scale.

Most debugging techniques are manual.

Need to design more automatic and scalable debugging tools
An error in a process propagates quickly to all processes

**MPI is widely used in large-scale HPC applications.**
Processes communicate among them to compute the solution of a problem.

**MPI processes are tightly coupled.**
A process needs to receive data from another process to make progress.

---

**Error propagation example**
```c
// computation code
for (...) {
    MPI_Send();
    // computation code
    for (...) {
        MPI_Recv();
        // computation code
    }
    MPI_Reduce();
    // computation code
    MPI_Barrier();
}
```

- Error here
- Some processes wait here
- All other processes wait here

*Hangs and slow execution are common bug manifestations*
Finding the least-progressed (LP) task often helps to identify the root cause of bugs.

// computation code
for (...) {
    MPI_Send()
    // computation code
    for (...) {
        MPI_Recv()
        // computation code
    }
    MPI_Reduce()
    // computation code
    MPI_Barrier()
}

Error here in task 4

Some processes wait here

All other processes wait here

Tools to Analyze the Progress of Tasks

**STAT**
- Static analysis technique
- Temporal ordering of tasks
- Identifies loop order variables (LOV)
- But, not all LOV can be identified

**AUTOMADE**
- Probabilistic technique
- Captures control-flow via Markov model
- But, cannot infer progress dependencies within loops
PRODOMETER: tool for accurate progress dependence analysis

Loop-aware progress-dependence analysis
To determine least-progressed (LP) task in MPI programs

Same dynamic technique as AutomaDeD
- Each task is represented via a Markov model
- Progress dependencies are determined probabilistically

Improved accuracy when a failure occurs within a loop
Talk overview

1. Background
   - Control-flow history
   - Progress dep. graph

2. Model annotations
   - Loop analysis
   - Characteristic edge

3. HPC benchmarks
   - Fault injection
   - Case study

Progress-dependence Analysis

Loop-aware Analysis

Evaluation
Each MPI Task is Modeled as a Markov Model

Sample code

```c
foo() {
    MPI_gather( )
    // Computation code
    for (...) {
        // Computation code
        MPI_Send( )
        // Computation code
        MPI_Recv( )
        // Computation code
    }
}
```

Markov Model

Nodes represent execution state before and after MPI calls

- Gather call stack
- Create states in the model
The Progress Dependence Graph (PDG)

The PDG facilitates finding the origin of performance bugs. It shows dependencies between tasks to make progress on the code.

The LP task can be identified visually at the top of the graph.
How to identify progress dependence dynamically?

Point-to-Point Operations

// computation code...
MPI_Recv(..., task Y, ...)
// ...

- If X calls MPI_Recv, X depends on task Y
- Dependence can be obtained from call parameters

Collective Operations

// computation code ...
MPI_Reduce(...)
// ...

- Multiple implementations (e.g., binomial trees)
- A task can reach MPI_Reduce and continue
- A task could block waiting for another task (less progressed)
Progress dependencies are identified probabilistically

Sample Markov Model

- Task A
- Task B
- Task C
- Task D
- Task E
- Task F

Probability(3 -> 5) = 1.0
Probability(5 -> 3) = 0

Task C is likely waiting for task B
(A task in 3 always reaches 5)

C has progressed further than B
Progress dependencies cannot be identified within loops

Sample Markov Model

We need more accurate progress-dependence analysis
Markov models are annotated to include transition counts

1. Old Markov model only had transition probability. New models include loop edge transition counts.

2. Markov models from all tasks are merged. New edges may be created.

3. We use a binomial-tree reduction for scalable model merging. Complexity: $O(\log \#\text{models})$

PDG analysis is done in a single task.
Estimating loop iterations in the Markov model

Transition counts may belong to multiple nested loops.
Counts are the sum of several loops.

How do we distinguish the number of iterations per loop?
How do we identify the number of iterations per loop?

**Characteristic edge:**
*The edge that is not part of any other loop.*

We use the **backedge** as the characteristic edge for a loop.

**Assumptions**

1. A loop makes a transition on the backedge when it completes an iteration
2. Backedge is not shared with any other loop
3. Loops are reducible (e.g., code does not use “goto” statements)
Lexicographic comparison of nested loops

Lexicographical order: in the order from outer to inner loop

Task X has made more progress on the inner loop than task Y. But Task Y has made more progress on the outer loop. Task Y has made more progress.
Workflow of the tool

- Online Markov model creation
- Each task creates a model

- User visualizes PDG and least-progressed task(s)

MPI Application + PRODOMETER library

- Hang detection occurs
- All processes trigger the diagnosis analysis

- If tasks are within loops, use loop-aware analysis:
  1. Identify loops
  2. Identify characteristic edges
  3. Measure progress dependence based on loop iterations

Monitoring

Failure detection

Markov model merging

Loop-aware progress-dependence analysis

PDG

- Markov models are combined in a single model
- The rest of the analysis is done in one task
Talk overview

1. Progress-dependence Analysis
   - Background
   - Control-flow history
   - Progress dep. graph

2. Loop-aware Analysis
   - Model annotations
   - Loop analysis
   - Characteristic edge

3. Evaluation
   - HPC benchmarks
   - Fault injection
   - Case study
Fault injection in six HPC benchmarks

HPC benchmarks: AMG, LAMMPS, IRS, LULESH, BT, SP

We inject a hang in a:
• Random MPI process
• Random function call

We only inject inside loops.
HPC applications spend most of its time (>90%) inside loops.

Experimental runs use 128, 256, and 512 MPI processes.
Metrics to compare the performance of the tools

**Accuracy:**
The fraction of cases that a tool correctly identifies the Least-Progressed (LP) tasks

**Precision:**
The fraction of the identified LP tasks that are actually where the fault was injected

---

**Evaluated tools**
(A) PRODOMETER  
(B) AUTOMATEDD  
(c) STAT

---

**EXAMPLES**

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hang</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LP tasks given by the tool</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td></td>
<td>(2, 3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Is the tool accurate?</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Precision</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td>50%</td>
</tr>
</tbody>
</table>
**PRODOMETER** is more accurate and precise than **AUTOMADeD**

**PRODOMETER (PR)**, **AUTOMADeD (AU)**

### Accuracy

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>128 tasks</th>
<th>256 tasks</th>
<th>512 tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>AU</td>
<td>PR</td>
<td>AU</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>AMG</td>
<td>0.92</td>
<td>0.94</td>
<td>0.88</td>
</tr>
<tr>
<td>IRS</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>LULESH</td>
<td>0.90</td>
<td>0.90</td>
<td>0.92</td>
</tr>
<tr>
<td>BT</td>
<td>0.82</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>SP</td>
<td>0.94</td>
<td>0.92</td>
<td>0.92</td>
</tr>
</tbody>
</table>

### Precision

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>128 tasks</th>
<th>256 tasks</th>
<th>512 tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>AU</td>
<td>PR</td>
<td>AU</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>0.98</td>
<td>0.99</td>
<td>0.98</td>
</tr>
<tr>
<td>AMG</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>IRS</td>
<td>0.96</td>
<td>0.98</td>
<td>0.97</td>
</tr>
<tr>
<td>LULESH</td>
<td>0.97</td>
<td>0.98</td>
<td>0.94</td>
</tr>
<tr>
<td>BT</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>SP</td>
<td>1.00</td>
<td>0.87</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Accuracy in PRODOMETER is on average 93%, versus 64% in AUTOMADeD**

**Precision is always higher in PRODOMETER than in AUTOMADeD**

**Comparison to STAT**

- We applied STAT to cases where PRODOMETER succeeded
- STAT depends on identifying LOV (loop order variables)
- STAT failed to identify LOVs in several cases (33%)
- STAT takes a few minutes (while PRODOMETER takes seconds)
**PRODOMETER scales to thousands of MPI processes**

*It takes only a few seconds for PRODOMETER to perform the analysis with thousands of tasks*

*Application slowdown is between 1.3 and 2.4*
Bug manifested on new MPI version.
Hangs manifested at 32,768 MPI tasks.
We use a reproducer at 4,096 with PRODOMETER in the same machine.

PDG shows around 15 tasks blocked in an MPI gather operation.

The results suggested that the bug was located in the communication layer (MPI)
Later developers confirmed that the bug originated in the MPI library (in all-gather collective).

Progress-dependence graph (PDG)
Conclusions

Our tool can diagnose failures in large-scale HPC applications with high accuracy.

Our loop-aware technique is more accurate than state-of-the-art techniques.

On average PRODOMETER achieved over 93% accuracy and 98% precision.

The case study shows it diagnose an unknown non-deterministic bug, reproducible only at large scale.
Code has been released at github

https://github.com/scalability-llnl/