Accurate Application Progress Analysis for Large-Scale Parallel Debugging

PLDI 2014
June 10, Edinburgh, UK

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

Lawrence Livermore National Laboratory

PURDUE UNIVERSITY

LLNL-PRES-655481
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

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.

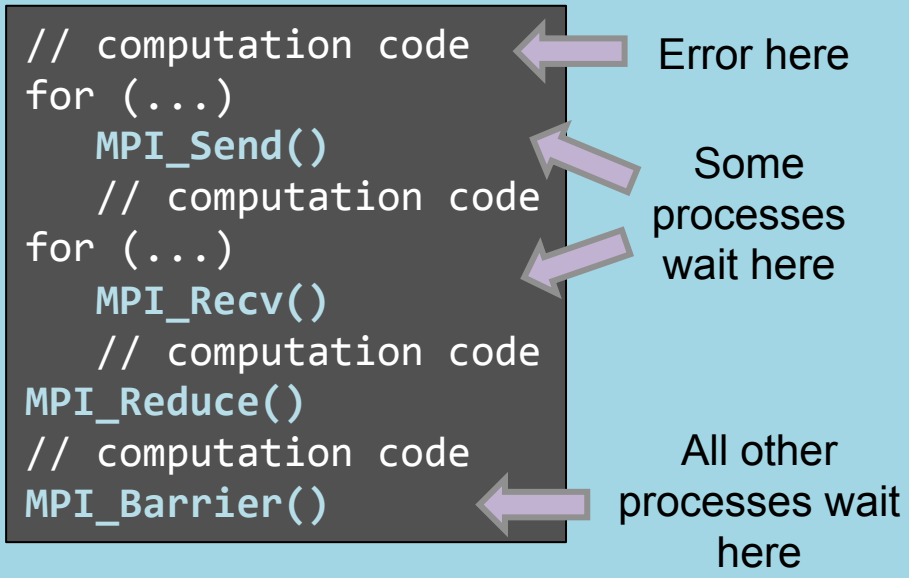
Need to design more automatic and scalable debugging tools

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.

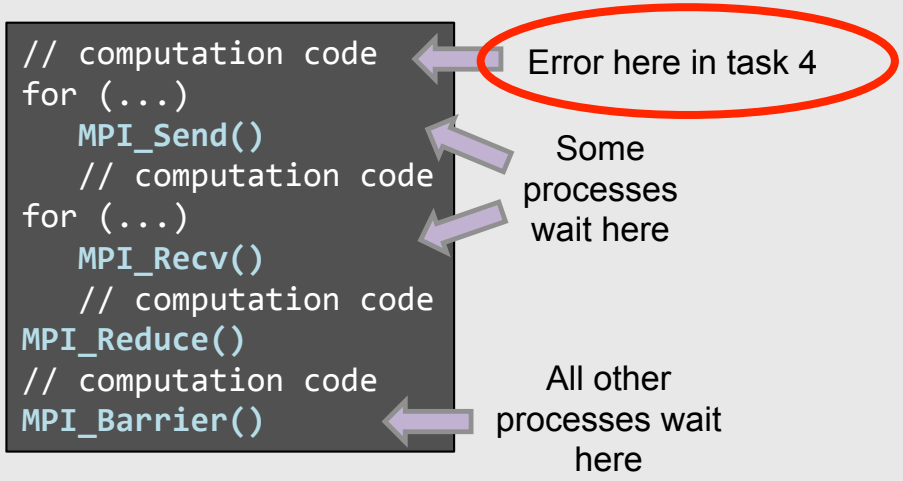
MPI processes are tightly coupled.
A process needs to receive data from another process to make progress.

Error propagation example



Hangs and slow execution are common bug manifestations

Finding the **least-progressed (LP) task** often helps to identify the root cause of bugs



Tools to Analyze the **Progress** of Tasks

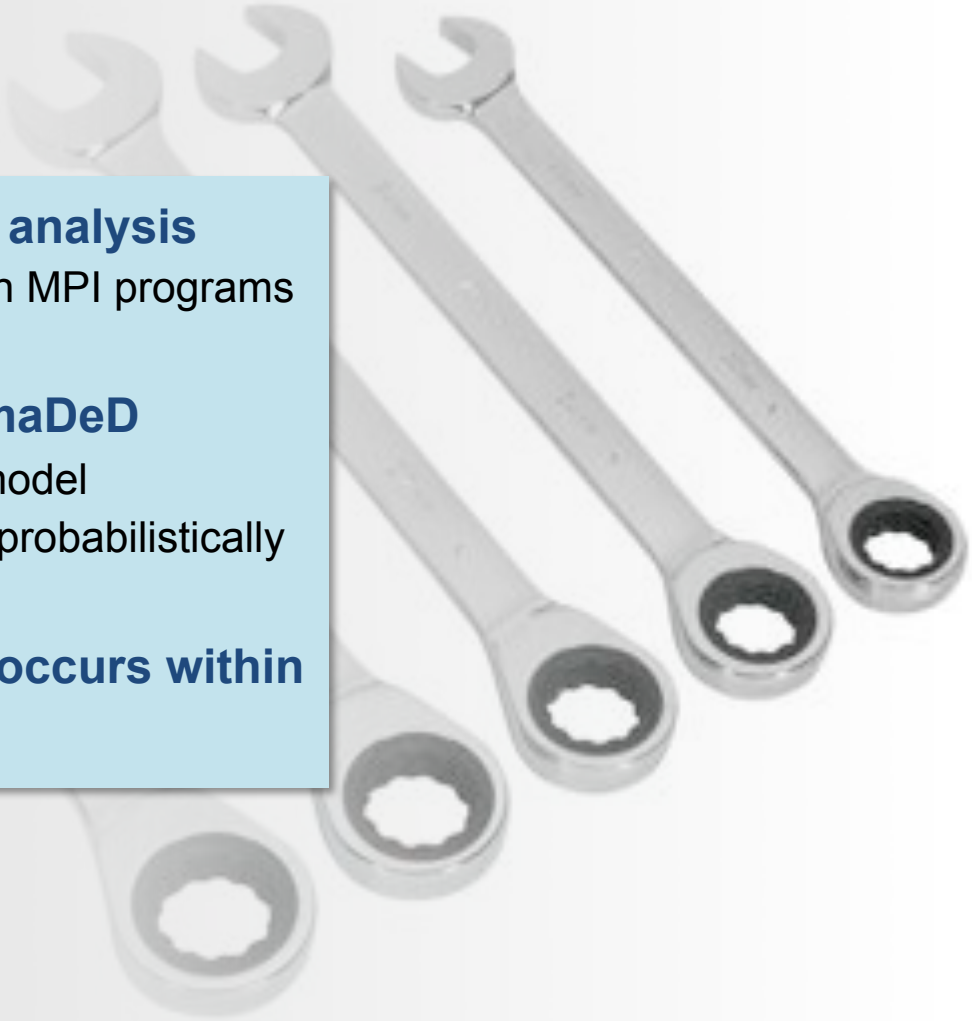
STAT

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

AutomaDeD

- 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

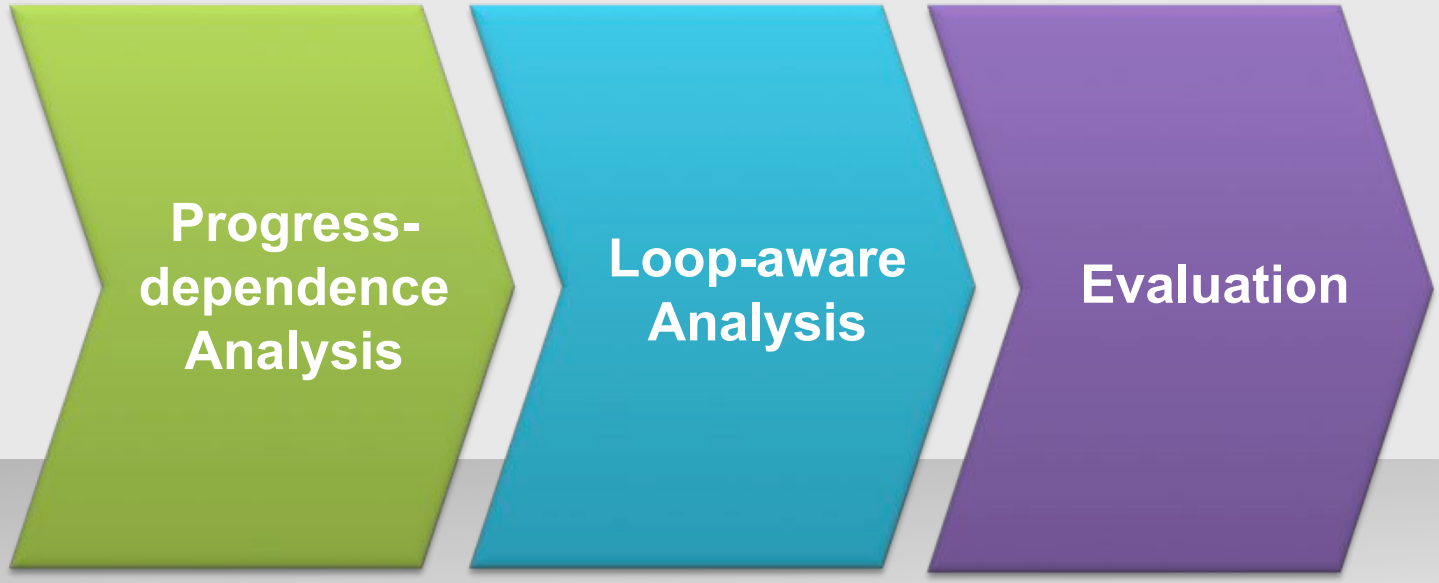
- ✚ Background
- ✚ Control-flow history
- ✚ Progress dep. graph

2

- ✚ Model annotations
- ✚ Loop analysis
- ✚ Characteristic edge

3

- ✚ HPC benchmarks
- ✚ Fault injection
- ✚ Case study



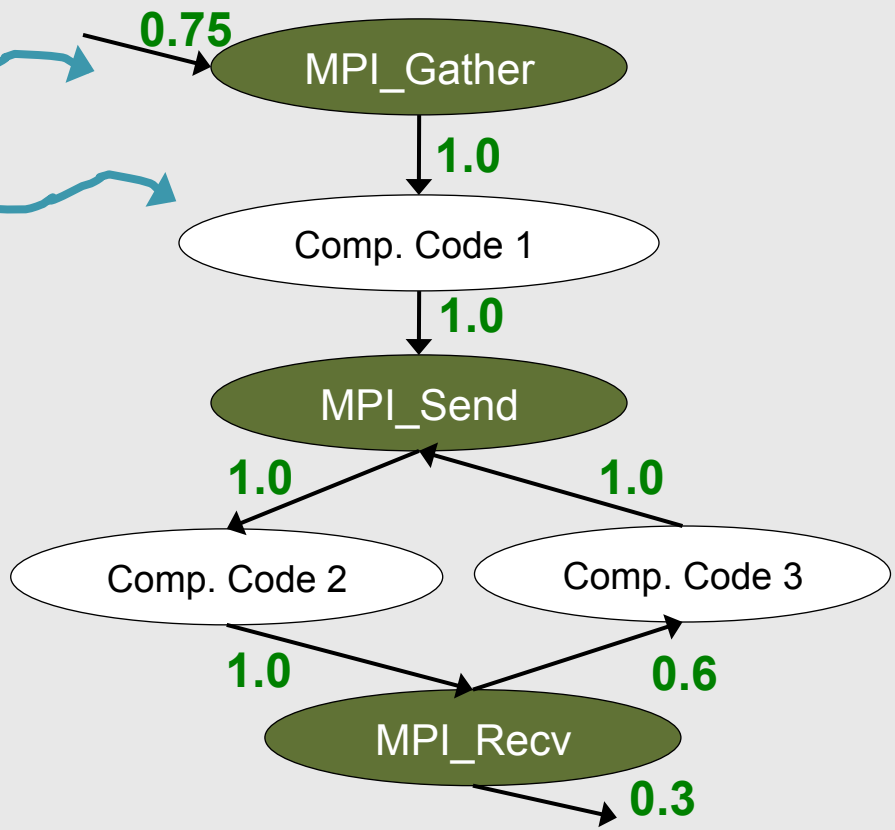
Each MPI Task is Modeled as a Markov Model

Sample code

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

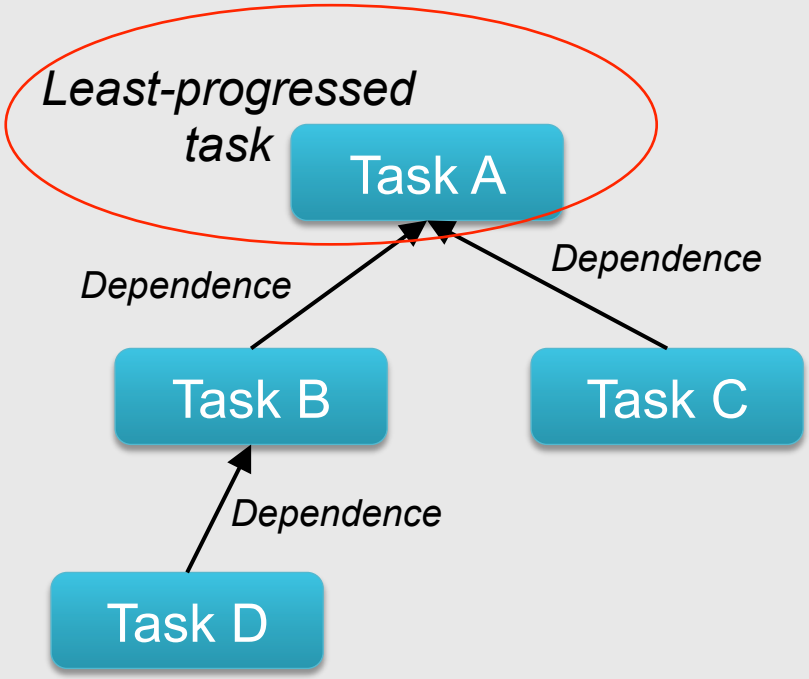
MPI calls wrappers:
- Gather call stack
- Create states in the model

Markov Model



Nodes represent execution state before and after MPI calls

The Progress Dependence Graph (PDG)



Introduced in AUTOMADED (PACT'12)

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

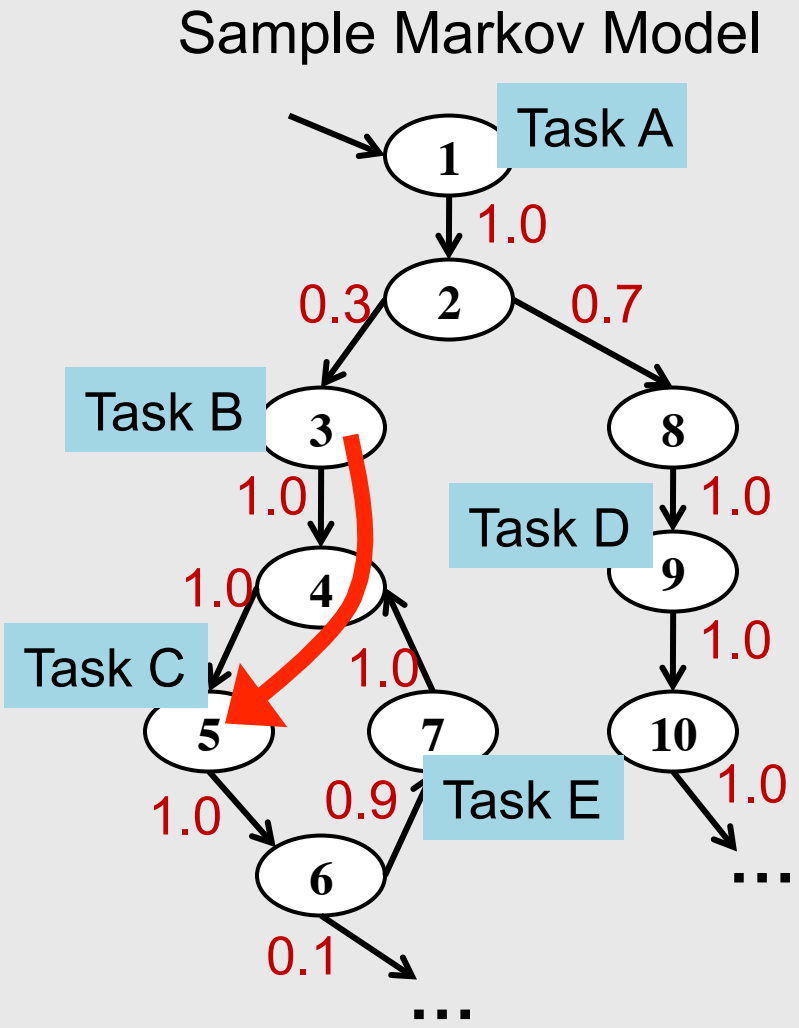
```
// computation code ...

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



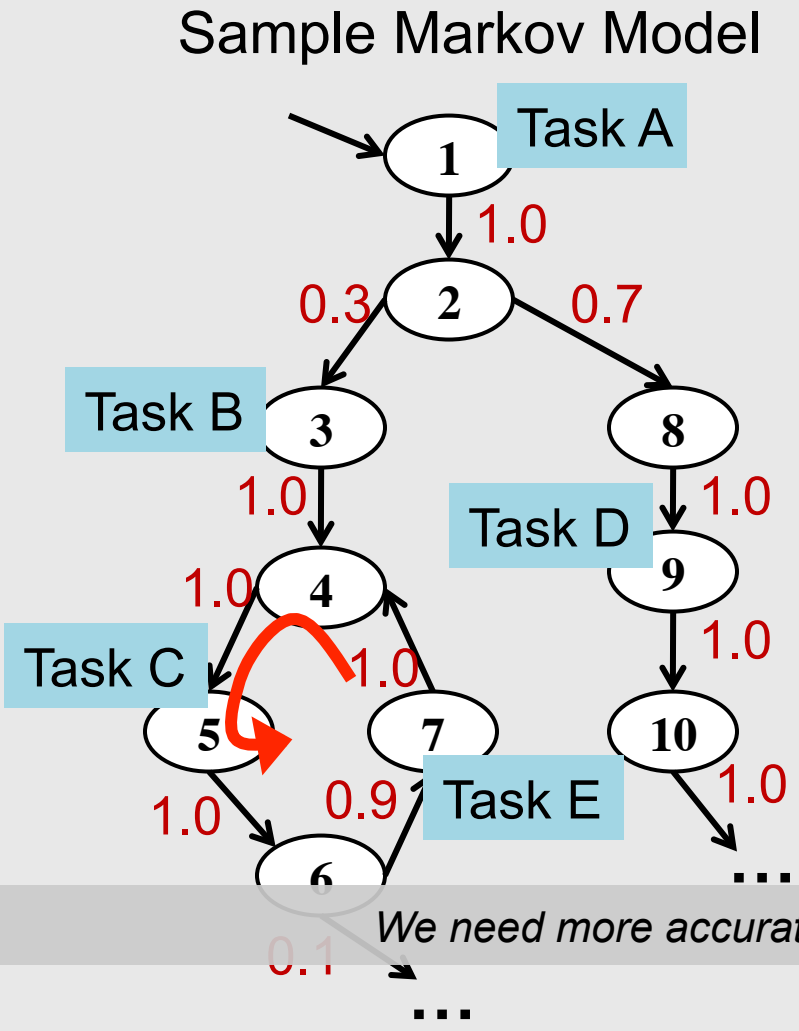
Progress dependence between tasks B and C?

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



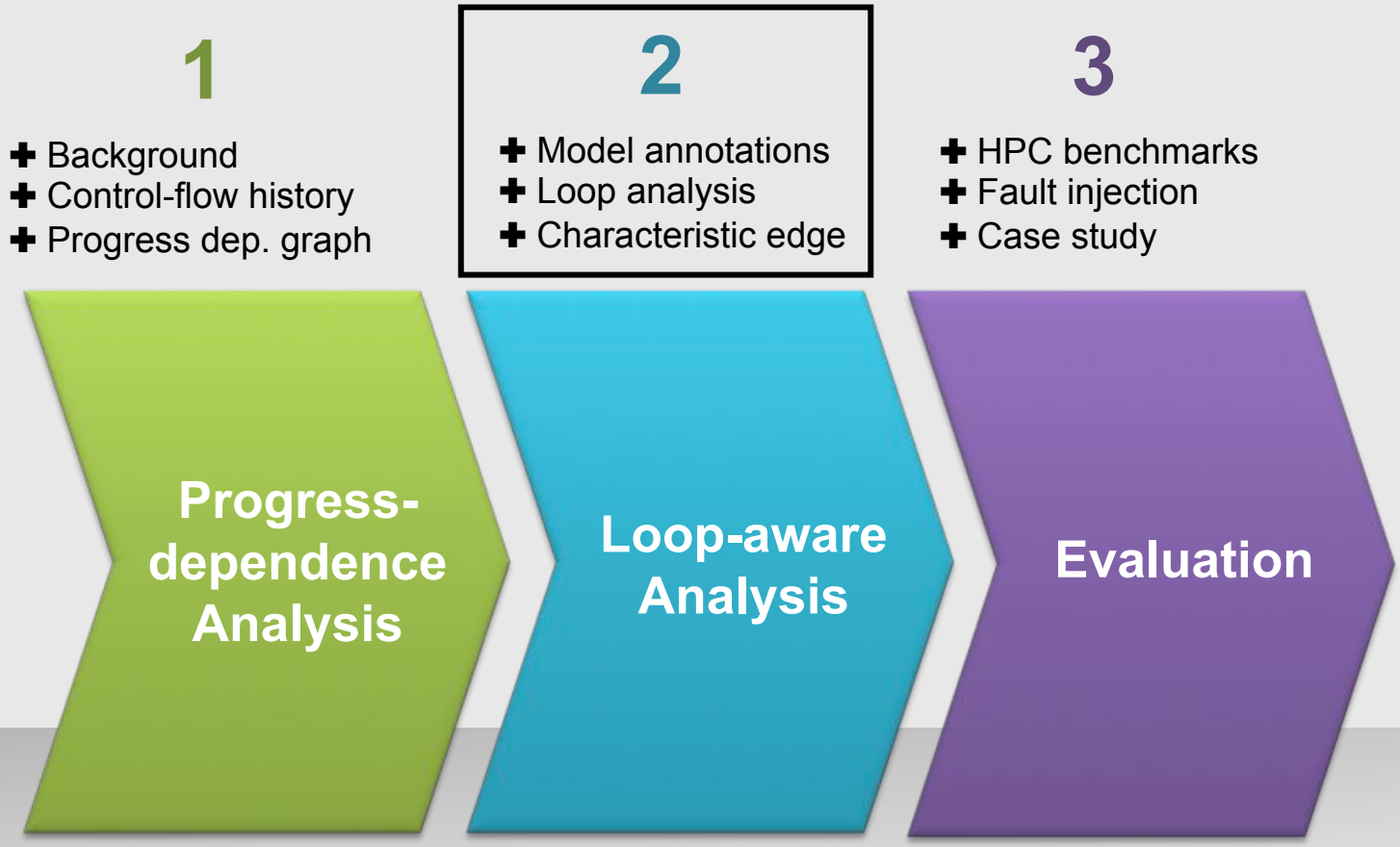
Dependence between tasks C and E?

Probability(7 → 5) = 1.0
Probability(5 → 7) = 0.9

What task has made more progress??

We need more accurate progress-dependence analysis

Talk overview

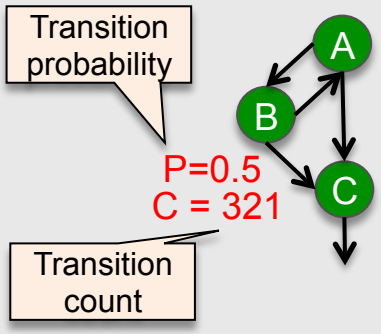


Markov models are annotated to include transition counts

1

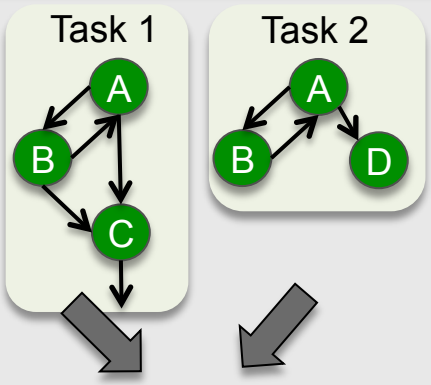
Old Markov model only had transition probability.
New models include loop edge transition counts.

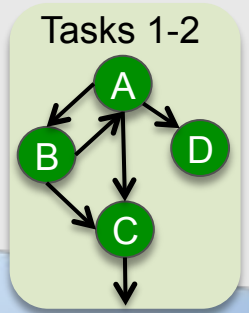
Markov model



2

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

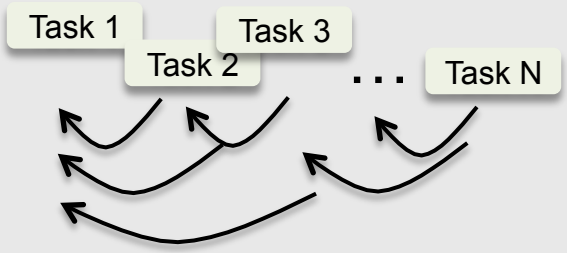




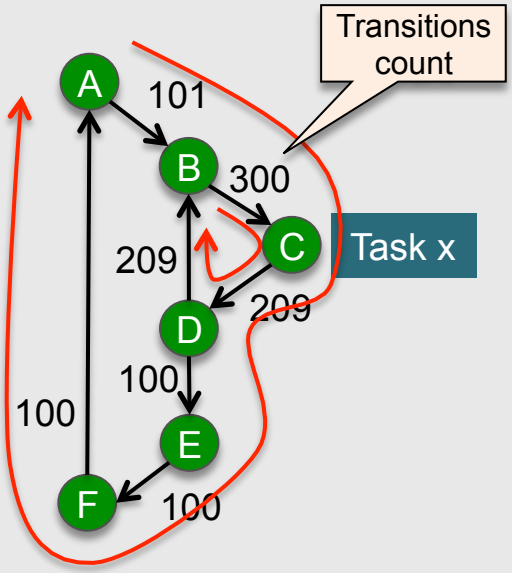
3

We use a binomial-tree reduction for scalable model merging.
Complexity: O(log #models)

PDG analysis is done in a single task.



Estimating loop iterations in the Markov model

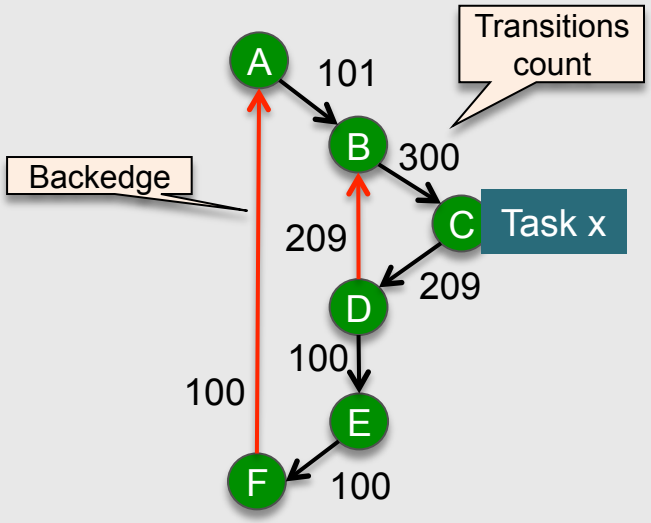


Transition counts may belong to multiple nested loops.

Counts are the sum of several loops.

How do we distinguish the number of iterations per loop?

Loop characteristic edge to identify loop iterations



How do we identify the number of iterations per loop?

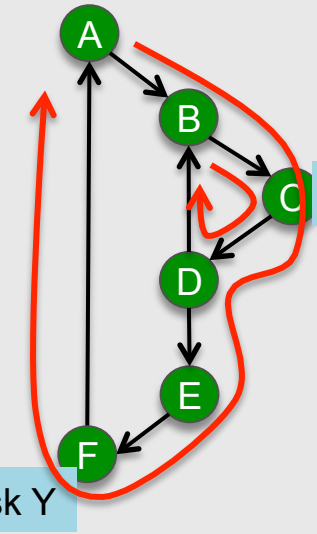
Characteristic edge:
The edge that is not part of any other loop.

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



Task X

Inner loop iterations = 200
Outer loop iterations = 100

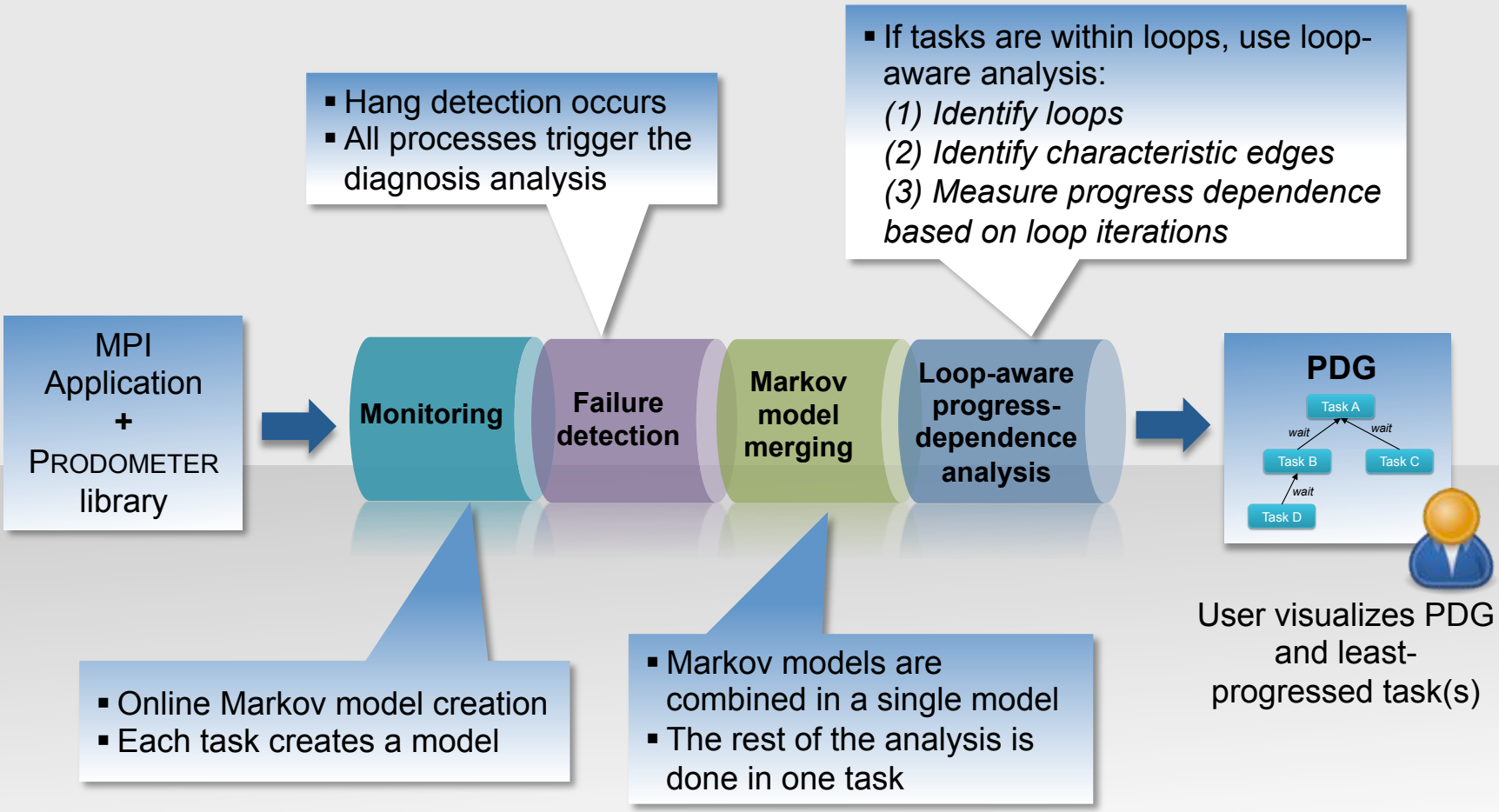
Inner loop iterations = 50
Outer loop iterations = 60

Task Y

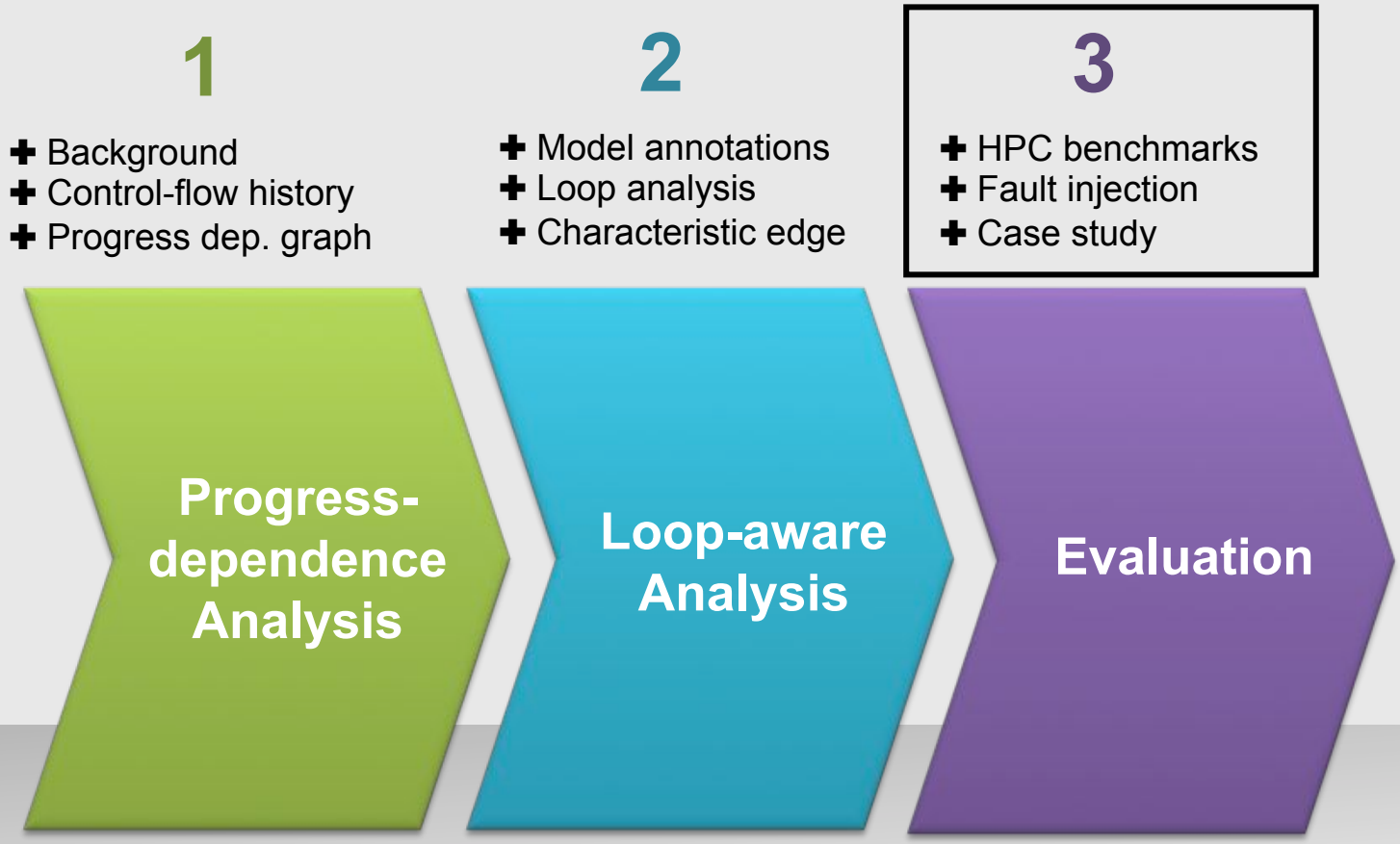
Lexicographical order: in the order from outer to inner loop

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.

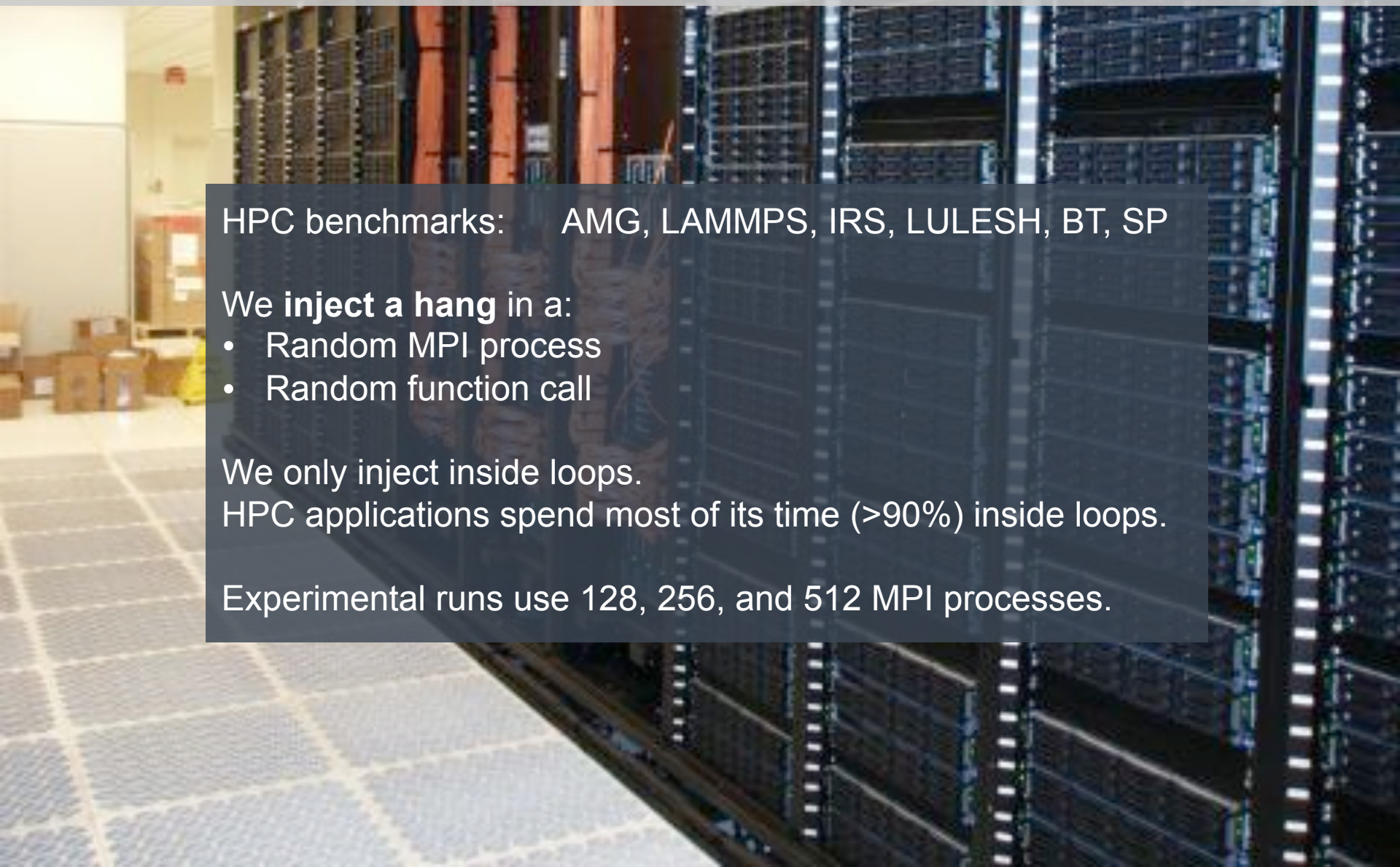
Workflow of the tool



Talk overview



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.

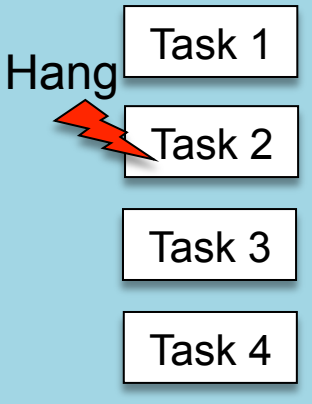
Metrics to compare the performance of the tools

Evaluated tools
(A) PRODOMETER
(B) AUTOMADED
(c) STAT

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

EXAMPLES



	Case 1	Case 2
LP tasks given by the tool	(2)	(2, 3)
Is the tool accurate?	Yes	Yes
Precision	100%	50%

PRODOMETER is more accurate and precise than AUTOMADED

PRODOMETER (PR), AUTOMADED (AU)

Accuracy

Benchmarks	128 tasks		256 tasks		512 tasks	
	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

Precision

Benchmarks	128 tasks		256 tasks		512 tasks	
	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

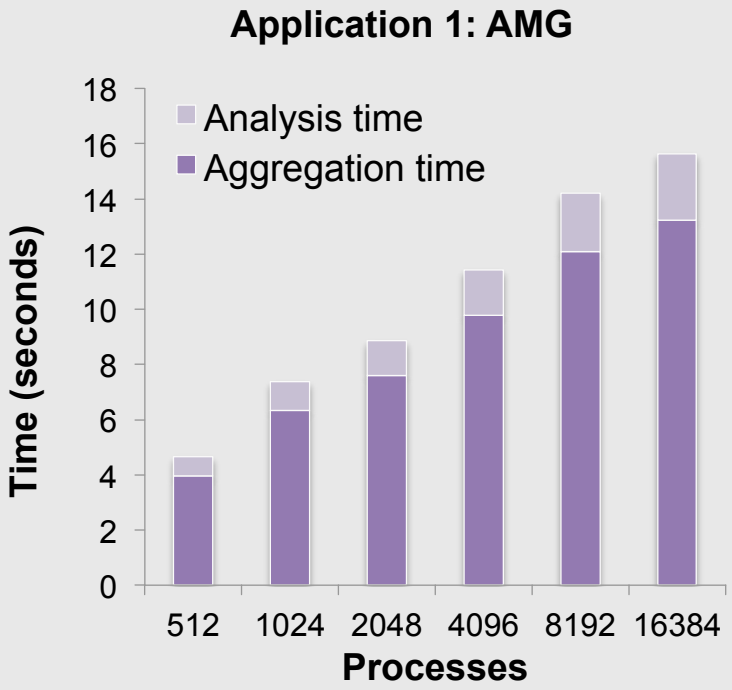
Accuracy in PRODOMETER is on average 93%, versus 64% in AUTOMADED

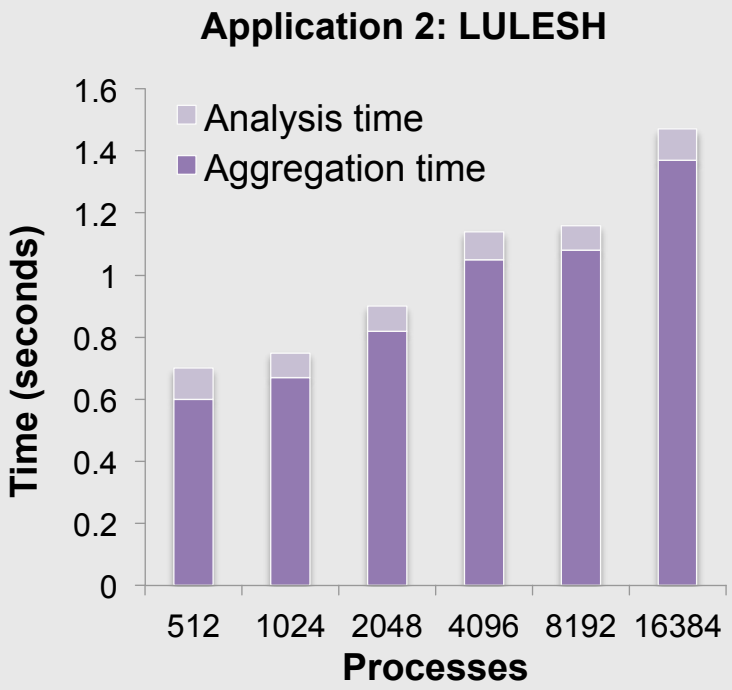
Precision is always higher in PRODOMETER than in AUTOMADED

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

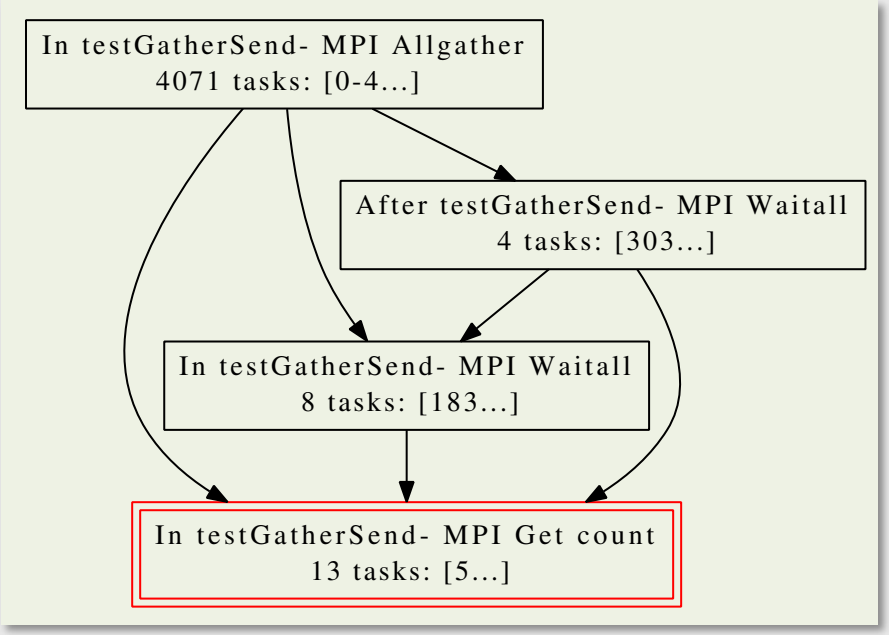
Case study on a real-world MPI bug

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.

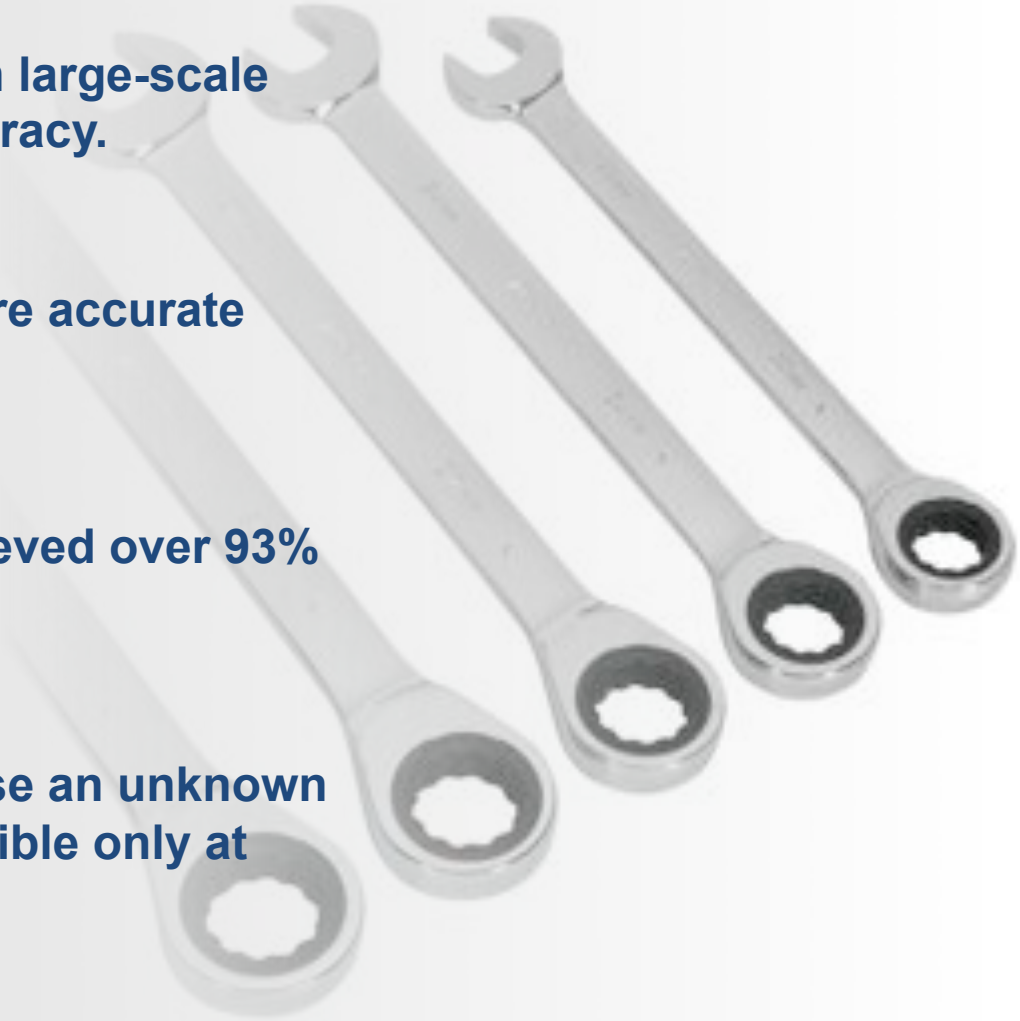
PDG shows around 15 tasks blocked in an MPI gather operation.

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**/