Debugging at Large Scales using "Triumph of Majority" Principle and Scaling Properties

Saurabh Bagchi
School of ECE, Purdue University

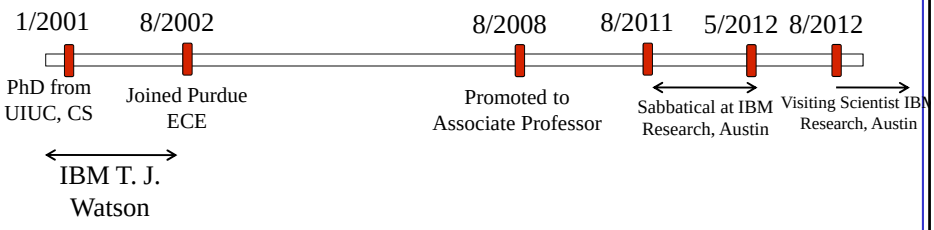
Joint work with: Ignacio Laguna, Nawanol Theera, Bowen Zhou, Milind Kulkarni (Purdue)
Greg Bronevetsky, Todd Gamblin, Bronis Supinski (LLNL)

Work supported by Department of Energy, National Science Foundation, Purdue Research Foundation

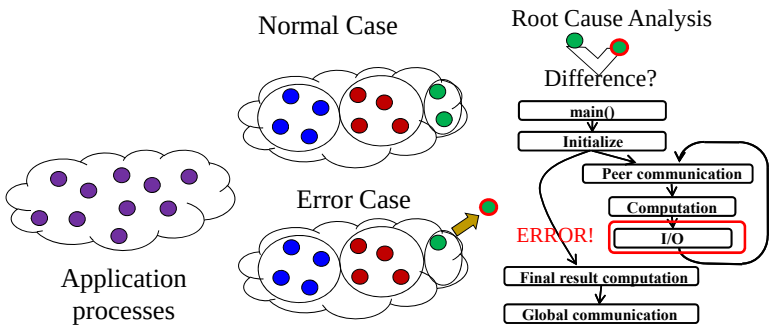
PURDUE UNIVERSITY

Time-line

- PhD (2001): University of Illinois at Urbana-Champaign
- Joined Purdue (2002)
- Promoted to Associate (2008)



Rapid detection and diagnosis of errors in large scale computer systems

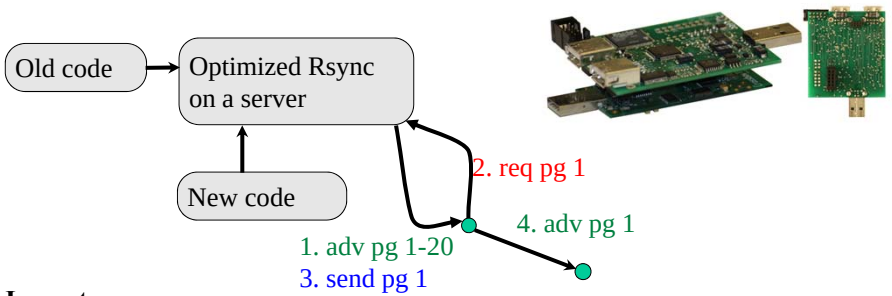


Tool operates in a distributed manner, scaling to the largest LLNL computing clusters (100,000+ processors - 2 orders of magnitude better scaling than prior work

Impact:

- Installed and used at LLNL, running on 100,000 processor Blue Gene cluster
- invitee to the DOE workshop (1/4 academics, all expenses paid) on failure resilience in exascale platforms (08/12); HPC Fellowship at SC '11

Debug, detect attacks, and efficiently update code on deployed wireless systems



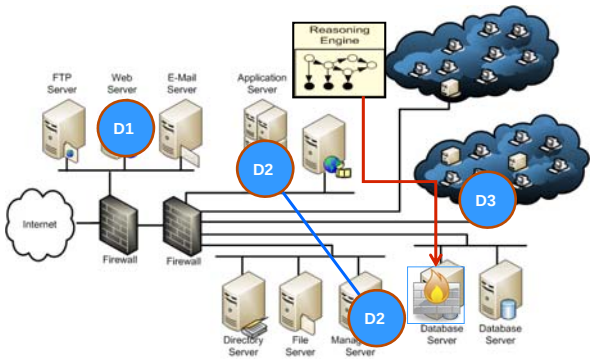
Impact:

Wireless reprogramming: Deployed in a city-wide wireless mesh network in South Bend, IN

Tracing: Only solution that provides complete tracing without slowing down embedded application (best paper, Sensys '11)

Security monitoring: His original work (DSN '05) started entire line of follow-on work – 156 citations – patent issued and licensed through Purdue

Rapid detection of, and response to, multi-stage security attacks against enterprise-class distributed systems



Impact

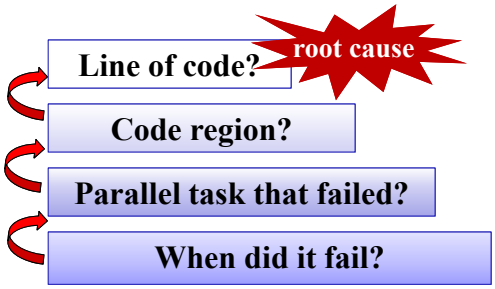
- Deployed at Northrup Grumman's internal cyber test range, 2+ years of funding
- Deployed at IBM Research on their Blade Centers (failures rather than attacks)
- Bayesian network formalism for inferencing based on sensor alerts was first in the field and has been picked up by HP Labs (w/Guy Lebanon)

Roadmap

- Detection of software (manifested) errors
  - Comparison of behavior models (DSN 10, SC 11)
  - Scale-dependent bugs (HPDC 11)
- Root cause analysis
  - Behavior modeling of application (PACT 12)
  - Scale-dependent bugs (HotDep 12)
- Insights and Open Questions

Developer Steps When Debugging a Parallel Application

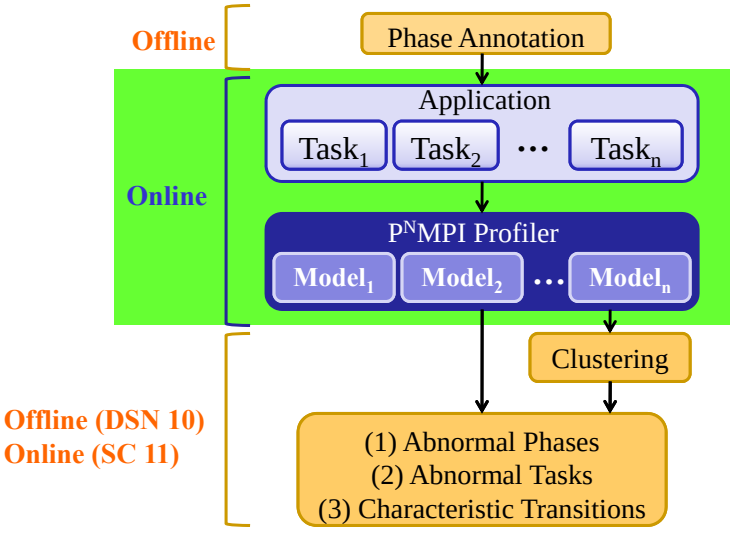
Questions a developer has to answer when an application fails:



- Need for tools to help developers find root cause quickly

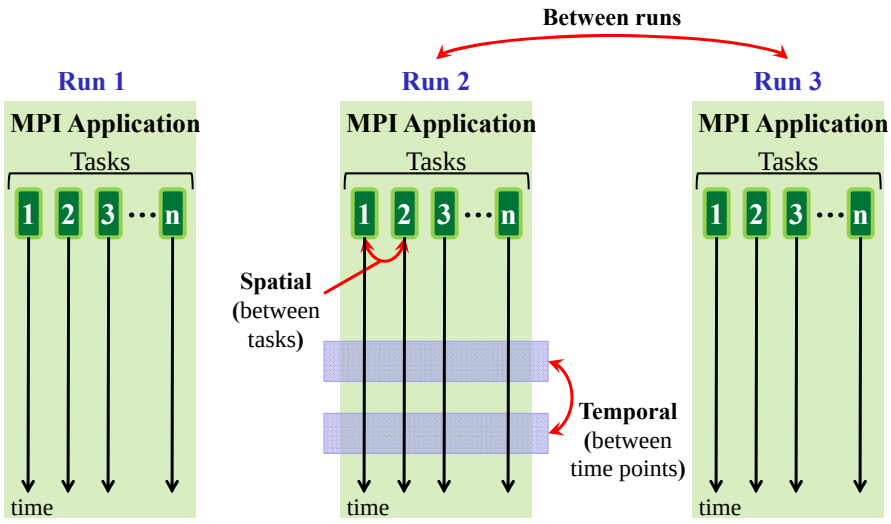
PURDUE UNIVERSITY

Our Error Detection & Localization Approach



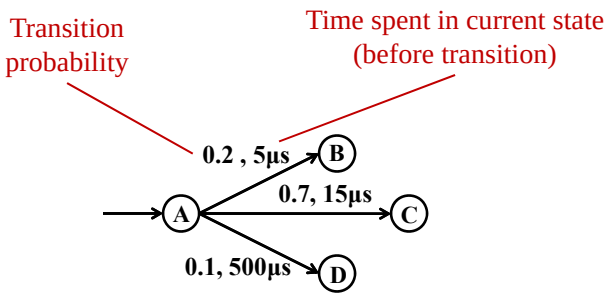
PURDUE UNIVERSITY

Types of Behavioral Differences



Semi-Markov Models (SMM)

- Like a Markov model but with *time* between transitions
  - Nodes: *application states*
  - Edges: *transitions from one state to another*



SMM Represents Task Control Flow

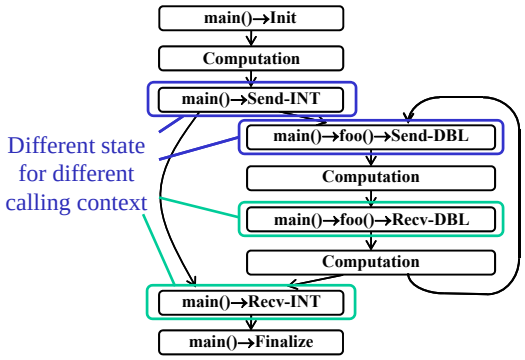
- States correspond to:
  - Calls to MPI routines
  - Code between MPI routines

Application Code

```
main() {
   MPI_Init()
   … Computation …
   MPI_Send(…, 1, MPI_INTEGER, …);
   for(…)
      foo();
   MPI_Recv(…, 1, MPI_INTEGER, …);
   MPI_Finalize();
}

foo() {
   MPI_Send(…, 1055, MPI_DOUBLE, …);
   …Computation…
   MPI_Recv(…, 1055, MPI_DOUBLE, …);
   …Computation…
}
```

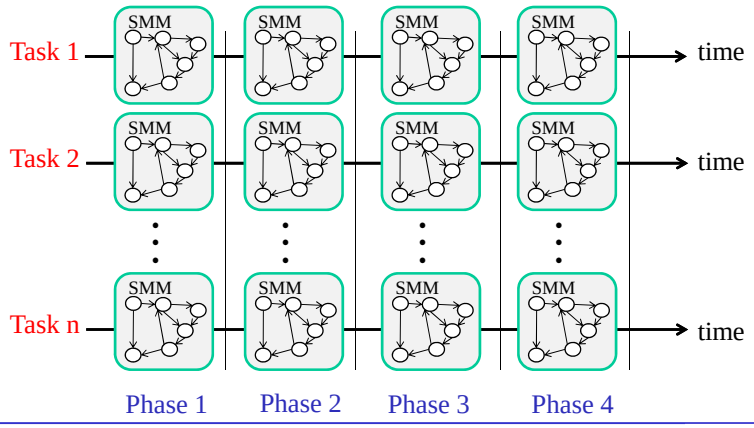
Semi-Markov Model



PURDUE UNIVERSITY

A Semi-Markov Model per Task, per Phase

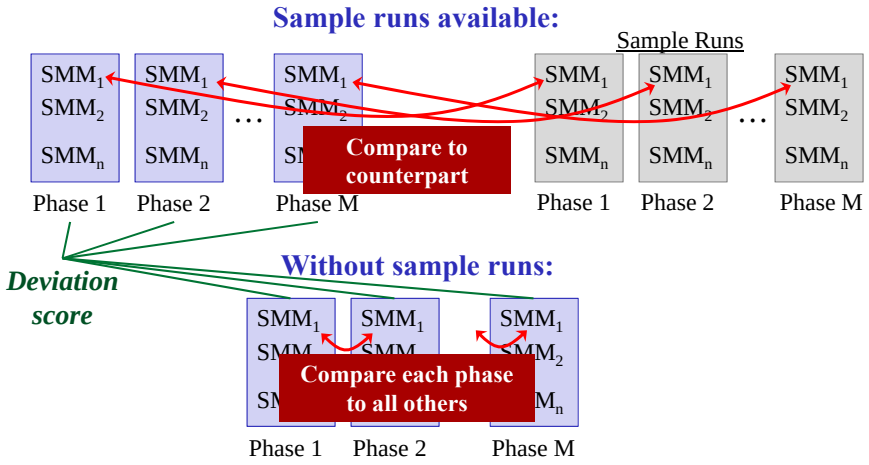
- Phases denote *regions of execution* within which behavior is statistically similar



PURDUE UNIVERSITY

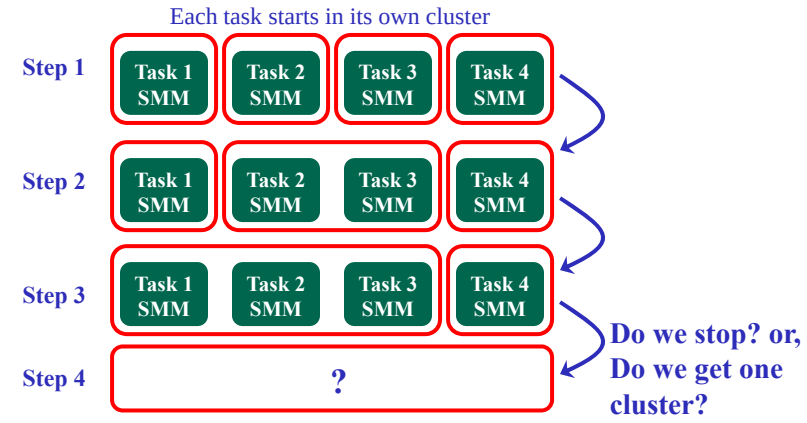
Faulty Phase Detection:
Find the Time Period of Abnormal Behavior

- **Goal:** *find phase that differs the most from other phases*



Clustering Tasks' Models:
Hierarchical Agglomerative Clustering (HAC)

Diss(SMM_1, SMM_2) = L2 Norm (Transition prob.) + L2 Norm (Time prob.)

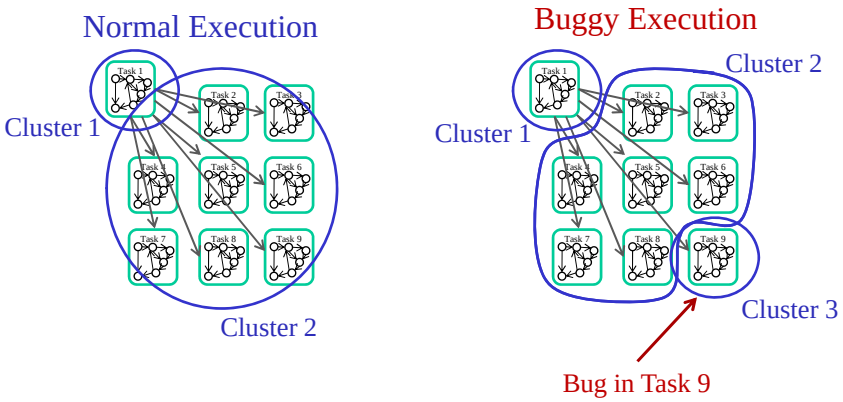


We need a dissimilarity *threshold* to decide when to stop

Cluster Isolation Example

Cluster Isolation: *to separate buggy task in unusual cluster*

Master-Worker Application Example

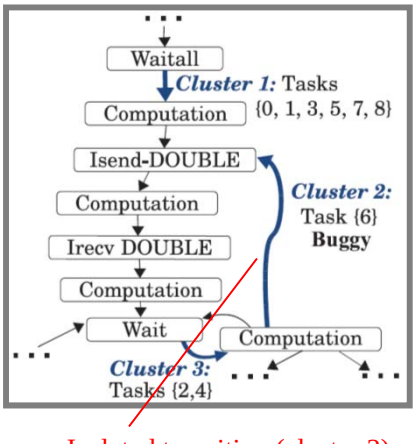


PURDUE UNIVERSITY

Transition Isolation: *Erroneous Code Region Detection*

- Method 1:
  - Find *edge* that distinguishes faulty cluster from the others
  - **Recall:** SMM dissimilarity is based in part on L2 norm of SMM's parameters
- Method 2:
  - Find *unusual individual edge*
  - Edge that takes unusual amount of time (compared to observed times)

Visualization of Results



Isolated transition (cluster 2)

PURDUE UNIVERSITY

Software Fault Model

- NAS Parallel Benchmarks (MPI programs):
  - BT, CG, FT, MG, LU and SP
  - 16 tasks, Class A (input)
- 2000 injection experiments per application:

Name	Description
FIN_LOOP	Local livelock/deadlock (delay 1,5, 10 sec)
INF_LOOP	Transient stall (infinite loop)
DROP_MESG	MPI message loss
REP_MESG	MPI message duplication
CPU_THR	CPU-intensive thread
MEM_THR	Memory-intensive thread

PURDUE UNIVERSITY

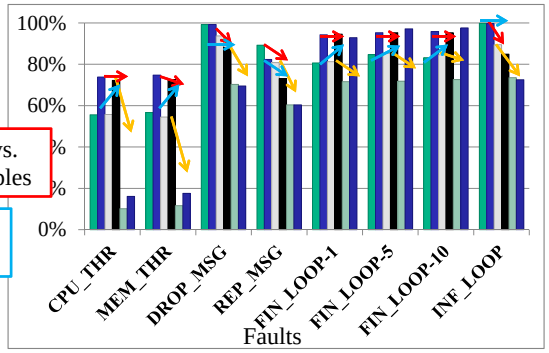
Phase Detection Accuracy

- ~90% for *loops* and *message drops*
- ~60% for *extra threads*
  - *Training* = sample runs available
  - Training significantly better than no training
  - Histograms better than Gaussians

Training vs. No Training

Some Faults vs. NoFault Samples

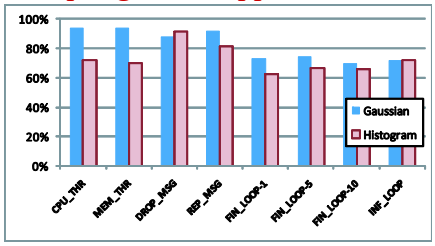
Gaussian vs. Histogram



PURDUE UNIVERSITY

Cluster Isolation Accuracy: Identifying Singleton Cluster

- Percentage of cases where the faulty task cluster consists of only a single task; Simplifies the problem of debugging because developer can focus on that single task
- Results for noisy sampling-based approach

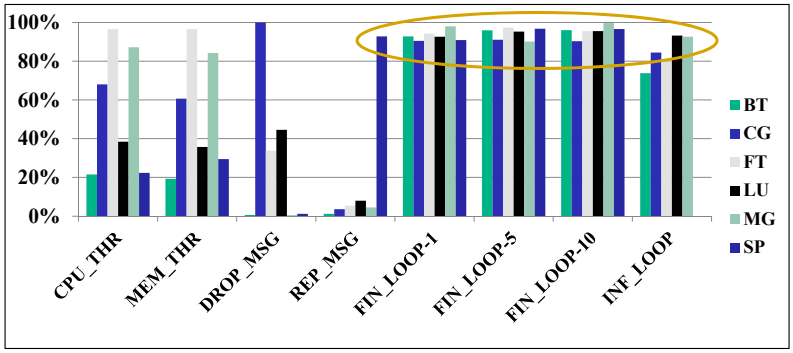


- Isolates the faulty task in more than 90% of cases for CPU_THR, MEM_THR, DROP_MESG, and REP_MESG
- Gaussian performs slightly better than Histograms due to greater sensitivity to outliers

PURDUE UNIVERSITY

Transition Isolation Accuracy

- Erroneous transition lies in top 5 candidates (identified by AutomaDeD)
  - Accuracy ~90% for loop faults
  - Highly variable for others
  - Less variable if event order information is used



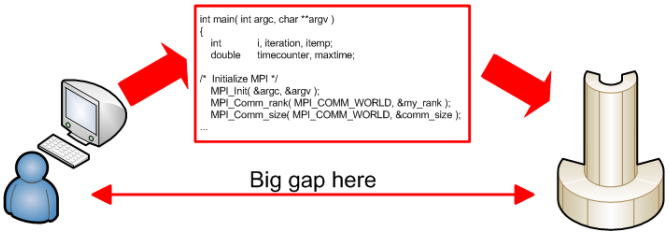
PURDUE UNIVERSITY

Roadmap

- Detection of software (manifested) errors
  - Comparison of behavior models (DSN 10, SC 11)
  - Scale-dependent bugs (HPDC 11)
- Root cause analysis
  - Behavior modeling of application (PACT 12)
  - Scale-dependent bugs (HotDep 12)
- Insights and Open Questions

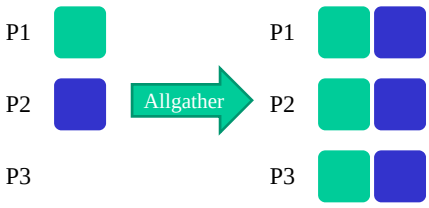
Scale-dependent Bugs in Parallel Programs

- What are they?
  - Bugs that arise often in large-scale runs while staying invisible in small-scale ones
  - Examples? Race condition, integer overflow, *etc.*
- Why do parallel programs have such bugs?
  - Coded and tested in small-scale development machines with small-scale test cases
  - Deployed in large-scale production systems



Example of Scale-dependent Bugs

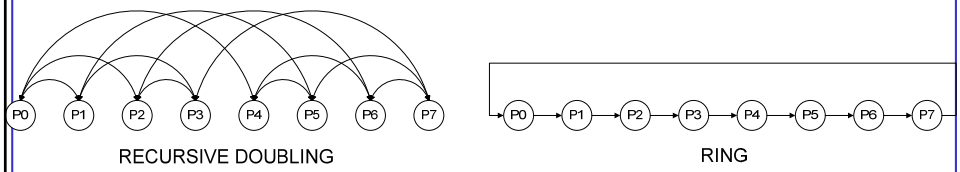
- A bug in MPI_Allgather in MPICH2-1.1
  - Allgather is a collective communication which lets every process gather data from all participating processes



PURDUE UNIVERSITY

Example of Scale-dependent Bugs

- MPICH2 uses distinct algorithms to do Allgather in different situations
- Optimal algorithm is selected based on the total amount of data received by each process



PURDUE UNIVERSITY

Example of Scale-dependent Bugs

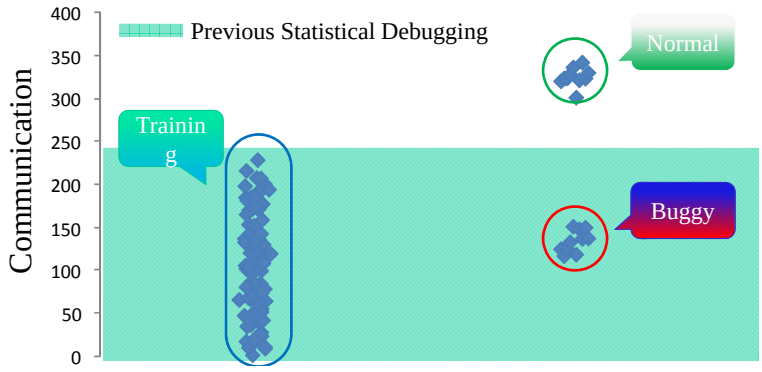
```
int MPIR_Allgather (
    ……
     int recvcount,
     MPI_Datatype recvtype,
     MPID_Comm *comm_ptr )
{
     int comm_size, rank;
     int curr_cnt, dst, type_size, le
    ……

     if ((recvcount*comm_size*type_size < MPIR_ALLGATHER_LONG_MSG) &&
         (comm_size is pof2 == 1)) {
         /* Short or medium size message and power-of-two no. of processes.
          * Use recursive doubling algorithm */
    ……
    else if (recvcount*comm_size*type_size < MPIR_ALLGATHER_SHORT_MSG) {
         /* Short message and non-power-of-two no. of processes. Use
          * Bruck algorithm (see description above). */
    ……
    else {  /* long message or medium size message and non-power-of-two
               * no. of processes. use ring algorithm. */
    ……
```

recvcount*comm_size*type_size can easily overflow a 32-bit integer on large systems and fail the if statement

PURDUE UNIVERSITY

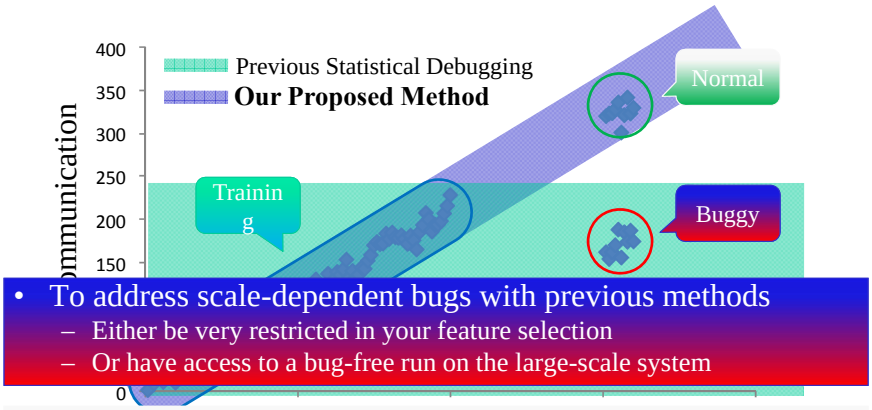
When Statistical Debugging Meets Scale-dependent Bugs



- Previous methods of statistical debugging for parallel programs
  [Mirgorodskiy SC'06] [Gao SC'07] [Kasick FAST'10]

PURDUE UNIVERSITY

When Statistical Debugging Meets Scale-dependent Bugs



- To address scale-dependent bugs with previous methods
  - Either be very restricted in your feature selection
  - Or have access to a bug-free run on the large-scale system
- Our method solved the problems of previous methods
  - Capable of modeling **scaling** behavior of parallel program
  - Only need access to bug-free runs from **small-scale** systems

PURDUE UNIVERSITY

Key Insights

- "**Natural scaling behavior**" of an application can be captured and used for bug detection
- Instead of looking for deviations from scale invariant behavior, we will look for deviations from the scaling trend

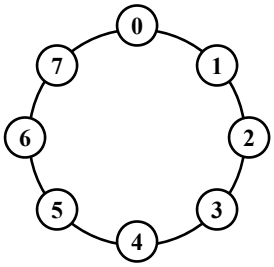
PURDUE UNIVERSITY

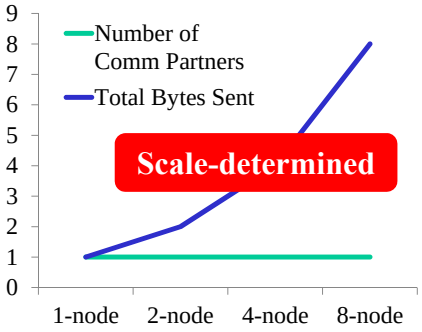
Contributions

- We build **Vrisha** to use scale as a parameter to build statistical model for debugging scale-dependent bugs
- **Vrisha** is capable of building a model to deduce the correlation between scale of run and program behavior
- **Vrisha** detects both application and library bugs with low overhead as validated with two real bugs from a popular MPI implementation

PURDUE UNIVERSITY

Key Observation

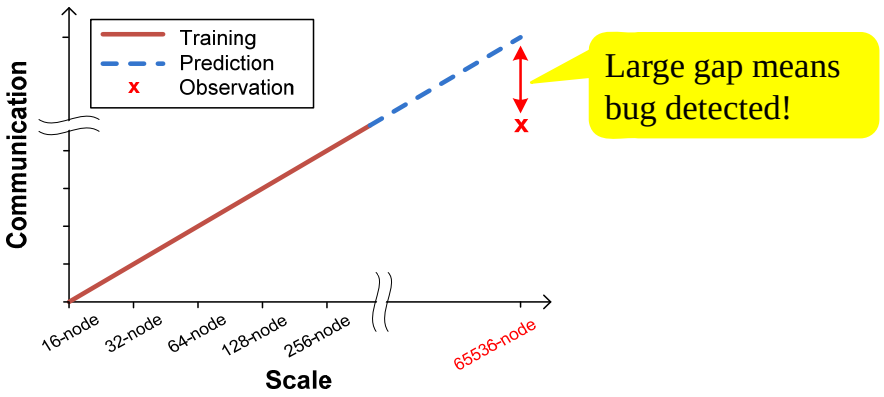




- We observe that some program properties are predictable by the scale of run in parallel programs
- These are called **scale-determined** properties

PURDUE UNIVERSITY

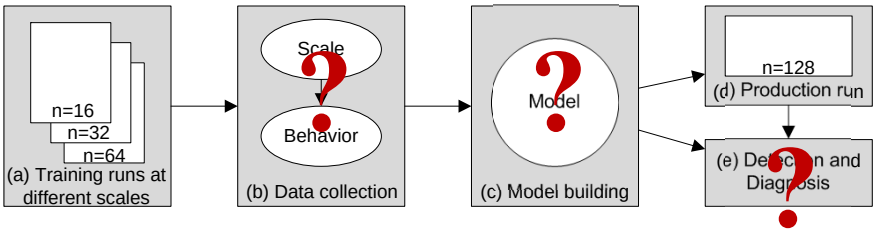
Bug Detection



Localization of bug is done by identifying the properties that cause the gap and the code regions that affect these properties

PURDUE UNIVERSITY

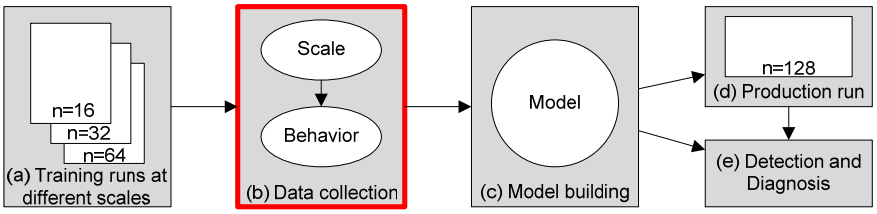
Vrisha's Workflow



a. Collect bug-free data from training runs of different scales
b. Aggregate data into scale and program property
c. Build a model from the scale and property
d. Collect data from the production run
e. Perform detection and diagnosis for the production run

PURDUE UNIVERSITY

Control Feature



- What features should we use to build the model of scaling behavior?
- We generalize the concept of "scale" to **control features**
- A set of parameters given by system or user
  - Number of processors in the system, Command-line arguments, Input size
- Control features are the predictors of program behavior

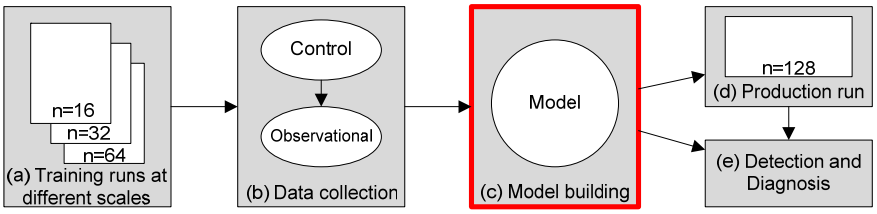
PURDUE UNIVERSITY

Observational Feature

- To characterize program behavior, we use **observational features**
- A set of vantage points in source code to profile various runtime properties
  - Example: MPI call point
  - Example: System call site
- Observational features are the manifestations of program behavior
  - Example: Amount of data communicated at each communication library call
  - Example: Number of system calls of a particular kind

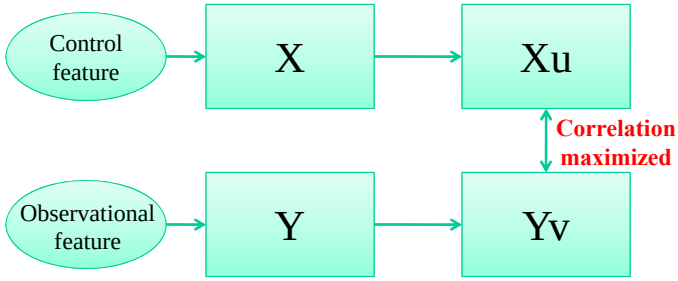
PURDUE UNIVERSITY

Challenges



- What model should we use to capture the relationship between scale and behavior?
  - Can describe both linear and non-linear relationships

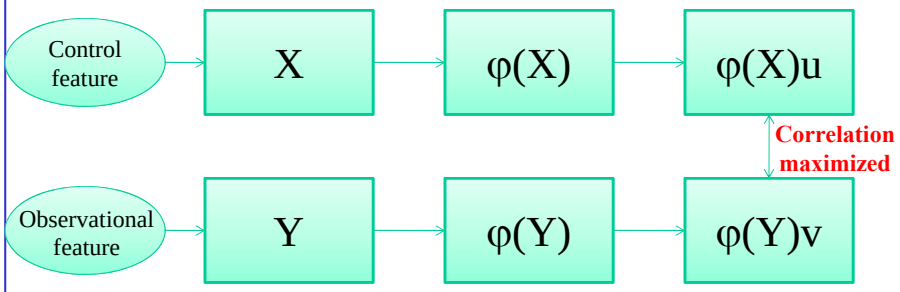
Model: Canonical Correlation Analysis



$$\max_{u,v} \quad corr(Xu, Yv)$$

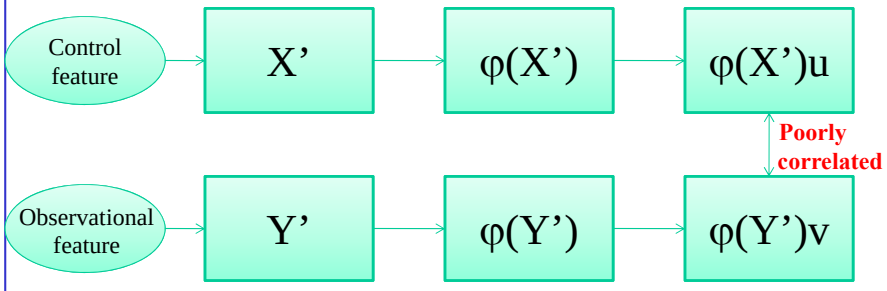
$$\text{such that} \quad \|u\| = \|v\| = 1$$

Model: Kernel CCA



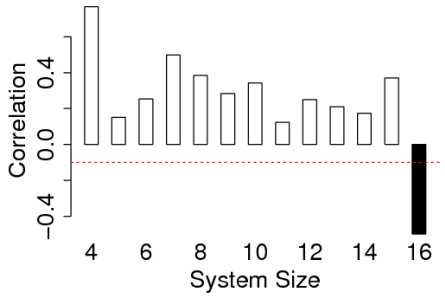
We use "kernel trick", a popular machine learning technique, to transform non-linear relationship to linear ones via a non-linear mapping φ(.)

Bug Detection



We declare a bug in the production run if the correlation between the control feature *X'* and the observational features *Y'* in the subspace defined by *u* and *v* is below certain range

Detect the Bug in Allgather



- The bug is configured to be triggered at 16-node run only
- A KCCA model is built solely on 4- to 15-node runs
- Vrisha is capable of revealing this scale-dependent bug with the very low correlation in the 16-node buggy run

PURDUE UNIVERSITY

Roadmap

- Detection of software (manifested) errors
  - Comparison of behavior models (DSN 10, SC 11)
  - Scale-dependent bugs (HPDC 11)
- Root cause analysis
  - Behavior modeling of application (PACT 12)
  - Scale-dependent bugs (HotDep 12)
- Insights and Open Questions

PURDUE UNIVERSITY

Why is Root Cause Analysis Hard?

- What error detection does *in the best case* is point out where the fault first manifests as an error
- Root cause analysis is supposed to point the developer to one or a small number of code regions
- This is difficult because of the distance between the fault and the failure
  - Particularly true for performance faults where fault in a different code region or even different process affects you

PURDUE UNIVERSITY

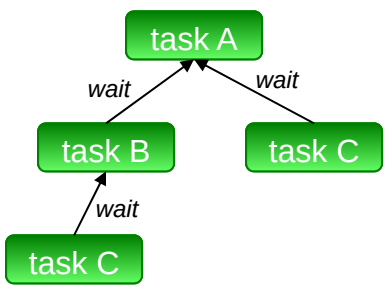
One Approach for Root Cause Analysis of Performance Faults

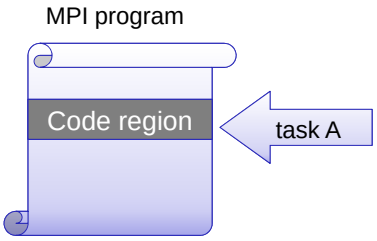
- Applicable to any application with communicating processes

1. Create a dependency graph that provides progress-dependence information: what task (or task group) prevents others from making progress?
2. From the dependency graph, determine the state and task group with the least progress
3. Apply backward slicing to determine code that influenced the least progressed task

PURDUE UNIVERSITY

The Progress Dependence Graph





- Facilitates finding the origin of performance faults
- Allows programmer to focus on the origin of the problem:
  - The *least progressed task*

PURDUE UNIVERSITY

What Tasks are Progress Dependent On Other Tasks?

Point-to-Point Operations

Task X:

```
// computation code...

MPI_Recv(…, taskY, …)

// ...
```

- Task X depends on task Y

- Dependency can be obtained from MPI calls parameters and request handlers

Collective Operations

Task X:

```
// computation code ...

MPI_Reduce(…)

// ...
```

- Multiple implementations (e.g., binomial trees)

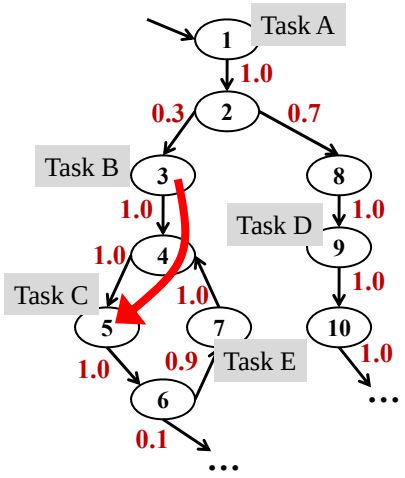
- A task can reach MPI_Reduce and continue

- Task X could block waiting for another task (less progressed)

PURDUE UNIVERSITY

Probabilistic Inference of Progress-Dependence Graph

Sample Markov Model



Progress dependence between tasks B and C?

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

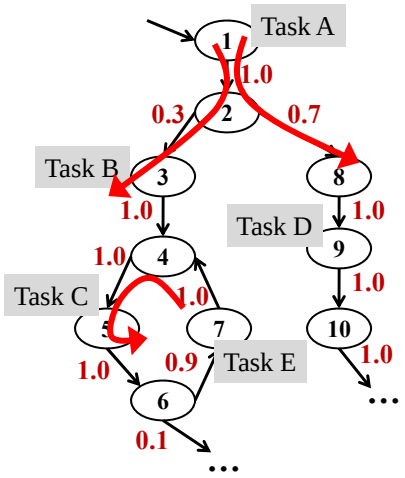
Task C is likely waiting for task B
(A task in 3 always reaches 5)

C has progressed further than B

PURDUE UNIVERSITY

Resolving Conflicting Probability Values

Sample Markov Model



Dependence between tasks B and D?

Probability(3 -> 9) = 0
Probability(9-> 3) = 0
The dependency is null

Dependence between tasks C and E?

Probability(7 -> 5) = 1.0
Probability(5-> 7) = 0.9

Heuristic: Trust the highest probability

C is likely waiting for E

PURDUE UNIVERSITY

Guided Application of Backward Program Slicing

- *Program slicing* identifies only statements that are relevant to a particular variable or statement.
- For root cause analysis, we only care about statements that could have led to the failure
  - Message-passing programs complicate program slicing
- Static or dynamic slicing is possible
- We start with the least progressed task in PDG, apply static slicing to it, provide code regions to developer in descending order of priority
- Result: Within 3 slices, the root cause is identified where the fault model is process slowdowns
- Result: Precision is 86% (43 out of 50 injections identified a singleton process)

Example of Backward Program Slicing

```
dataWritten = 0
for (…) {
   Probe(…, &flag,…)
   if (flag == 1) {
      Recv()
      Send()
      dataWritten = 1
   }
   Send()
   Recv()
   // Write data
}
if (dataWritten == 0) {
   Recv()
   Send()
}
Reduce()
Barrier()
```

Least-progressed task State

Shows the program slices that dataWritten is dependent on

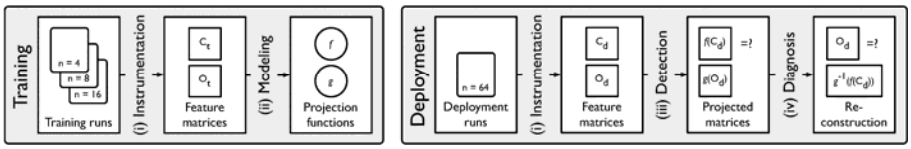
Roadmap

- Detection of software (manifested) errors
  - Comparison of behavior models (DSN 10, SC 11)
  - Scale-dependent bugs (HPDC 11)
- Root cause analysis
  - Behavior modeling of application (PACT 12)
  - Scale-dependent bugs (HotDep 12)
- Insights and Open Questions

PURDUE UNIVERSITY

Root Cause Analysis for Scale-Dependent Bugs

- Recall detection strategy for scale-dependent bugs: Vrisha
- Abhranta does root cause analysis
  - It predicts what the behavior of the buggy program *would have been* (according to the model) had the bug not occurred
- Insights for root cause analysis:
  - Even though the observational features for a buggy program, O_d are anomalous, the control features for the program, C_d, are nevertheless correct
  - A good guess for the correct observation features is an O_d' such that $g(O_d')$ is correlated with $f(C_d)$

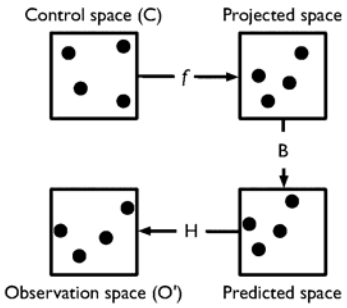


PURDUE UNIVERSITY

How Do We Invert The Function?

- Intuitive way of computing the notional correct observational features is by $o' = g^{-1}(f(c))$
- Trouble is that the function g is not invertible
  - In essence, g is an infinite degree polynomial
- Solution: Use a linear projection function for g, while leaving f as a non-linear function
  - f remains an infinite-degree polynomial, allowing us to still model program behaviors that vary polynomially with scale
  - Empirically determine that the choice of linear function for g does not hamper detectability of the system

Projections for Root Cause Analysis



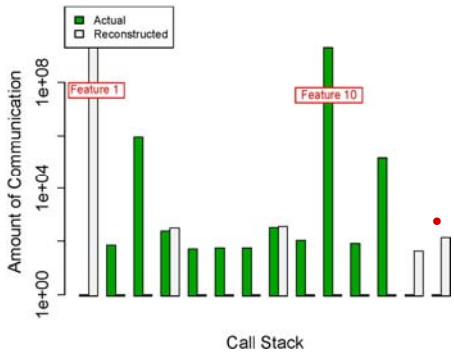
- The nominal observation space features are computed as:

$$O' = H.B.f(C)$$

- How to compute **H** and **B**? See our HotDep 12 paper

Case Studies for Root Cause Analysis

- Recall the bug in MPI_AllGather that caused a performance sub-optimal path to be taken for large scales



- ABHRANTA ranks all the observational features in descending order of reconstruction error (difference between ***O'*** and ***O***)
  - This is the suggested order of examination to find the bug
- The call stacks of the top two features, Features 1 and 10, differ only at the buggy if statement inside ALLGATHER, precisely locating the bug

PURDUE UNIVERSITY

Take-Away Lessons

- Contributions:
  - Novel way to model and compare parallel tasks' behavior
  - Focus debugging efforts on time period, tasks and code region where bug is first manifested
  - Scalable technique
  - For a subtle class of bugs called scale-dependent bugs, a technique that estimates scale-determined properties
  - Root cause analysis using program slicing and machine learning techniques
- Insights:
  - Different machine learning techniques should be part of our toolkit
  - Different kinds of bugs need different approaches: modeling, detection, root cause analysis
  - Detection and localization are significantly aided by training runs and developer hints
  - Detection techniques need to be online, and hence scalable; root cause analysis can be offline, but precise

PURDUE UNIVERSITY

Open Questions

- What should be our observation points and observation variables?
  - Depends on the kinds of bugs that are anticipated
  - Technique should be able to handle supersets
- How do we handle non-determinism in the execution environment?
  - Due to interfering processes on the same host
  - Due to network conditions
  - Also heterogeneity in execution environment
- How to handle data-dependent bugs
  - Need to characterize the data dependence
  - Does it depend on structure of data or also data values?

Backups

How to Make these Techniques Scalable?

- Goal: To run the detection and localization techniques online at a large scale
- We have developed three techniques:
  - Efficient edge comparison
  - Compress graphs of SMMs
  - Scalable outlier detection
- With all the three techniques in place, time for complete analysis (overhead of compression, detection, and localization) is 8.55 seconds at 10K tasks

Efficient Edge Comparison

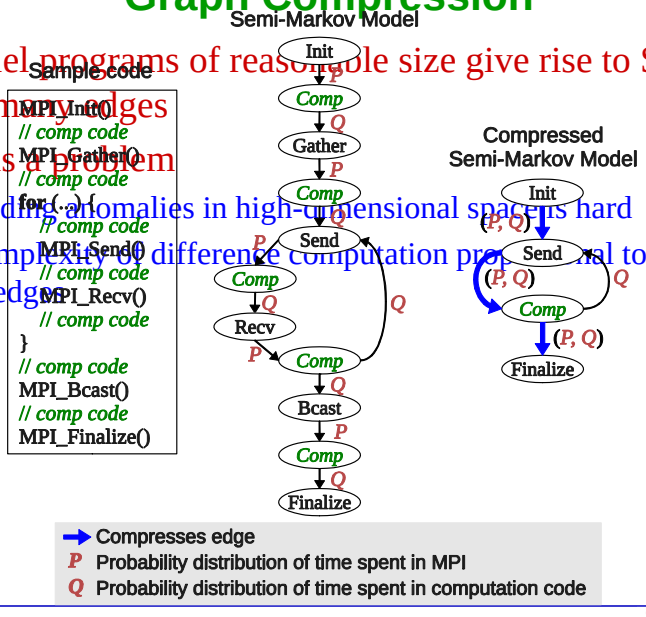
- Does the same edge exist in both graphs?
  - Compare two nodes – the source and the destination of the edge
  - Each node is represented by a call stack path
  - A method to compare call stack paths by reference
- What is the difference in the time distributions on the edges?
  - Exact computation is LK norm of two continuous distributions

$$\int_{-\infty}^{\infty} |P(x) - Q(x)|^k dx$$

  - But for Normal distributions, the area of overlap can be calculated from a lookup table, given two means and standard deviations

Graph Compression

- Parallel programs of reasonable size give rise to SMMs with many edges
- This is a problem
  - Finding anomalies in high-dimensional spaces is hard
  - Complexity of difference computation proportional to number of edges



PURDUE UNIVERSITY

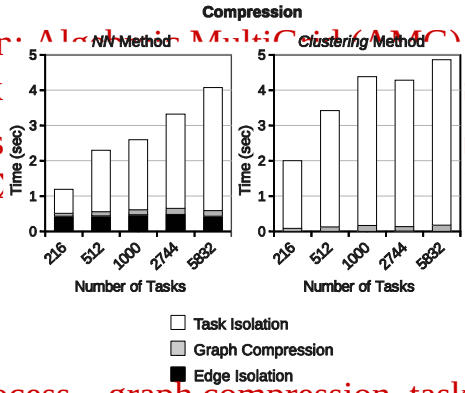
Scalable Outlier Detection

- Scalable clustering
  - K-medoids method
  - Traditional sequential clustering has linear or quadratic complexity in the number of points
  - To scale up, we use sampling with a fixed number of points
  - Scalable parallel reduction operation
  - Overall complexity is O(log n)
- Scalable nearest neighbor detection
  - Sample tasks using deterministic pseudo-random number generator
  - Find nearest neighbor among the sampled tasks
  - Need *k*-nearest neighbor if we believe up to *k-1* tasks can be affected by the fault

PURDUE UNIVERSITY

Performance Result with Scalability Techniques

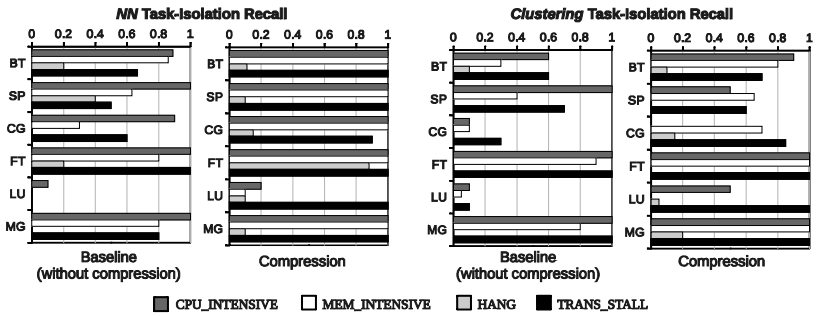
- Application: Algebraic MultiGrid (AMG) 2006 benchmark suite
- Cluster has [illegible]res with 2.8 GHz Intel [illegible] RAM



- Entire process – graph compression, task isolation, edge isolation – takes less than 5 seconds for both clustering and NN
- Graph compression incurs low overhead (< 170 ms), but improves runtime and error detection performance

PURDUE UNIVERSITY

Detection Coverage with Compression



- Compression improves accuracy in both NN and clustering methods
- Dimensionality reduction eliminates noisy dimensions from analysis
- What happens if multiple faulty tasks?

PURDUE UNIVERSITY