International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)
Seattle, Nov, 2011

Large Scale Debugging of Parallel Tasks with AutomaDeD

Ignacio Laguna, Saurabh Bagchi

PURDUE UNIVERSITY

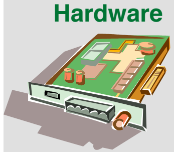
Todd Gamblin, Bronis R. de Supinski, Greg Bronevetsky, Dong H. Ahn, Martin Schulz, Barry Rountree

Debugging Large-Scale Parallel Applications is Challenging

- Millions of cores soon in largest systems
- Increased difficulty in developing correct HPC applications
- Poor scalability of traditional debuggers

Faults come from various sources

Hardware



- Physical degradation
- Soft / hard errors
- Performance faults

Software

- Coding bugs
- Misconfigurations

Fault Detection/Diagnosis is done Offline

Existing approaches
(*debuggers & statistical tools*)

Time

- **Fault manifestation**
- Stream traces to a **central component**
  - *from all processes*
- Analyze traces **offline** in central component
- **Pinpoint** where the problem is

Our approach

Time

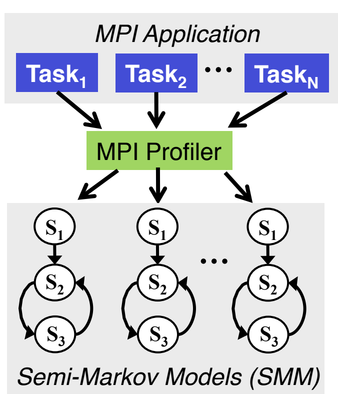
- **Fault manifestation**
- Analyze traces **online**
- **Pinpoint** where the problem is

How?
- Distributed / scalable methods
- Efficient data structures

PREVIOUS WORK

AutomaDeD

Fault detection & diagnosis in MPI applications (DSN '10)



Find offline:

phase → task → code region

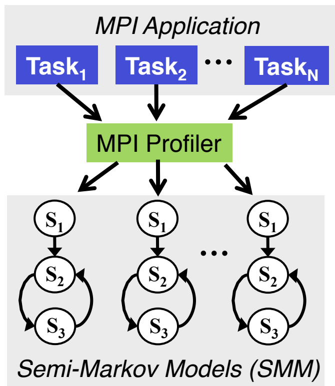
- Collect traces via MPI wrappers
  - Before and after call

```
MPI_Send(…) {
    tracesBeforeCall();
    PMPI_Send(…);
    tracesAfterCall();
    …
}
```

  - Collect call-stack info
  - Time measurements

PREVIOUS WORK

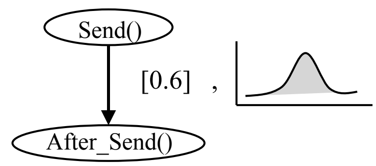
Modeling Timing and Control-Flow Structure



Find offline:

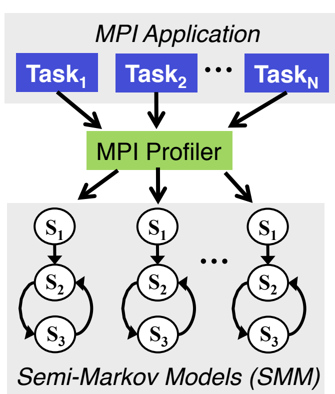
phase → task → code region

- States:
  - (a) code of MPI call, or
  - (b) code between MPI calls
- Edges:
  - Transition probability
  - Time distribution



PREVIOUS WORK

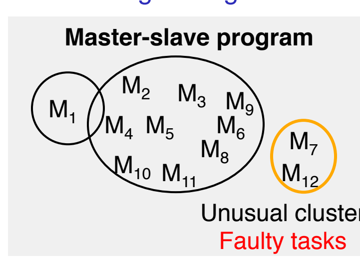
Detection of Anomalous *Phase* / *Task* / *Code-Region*



Find offline:

phase → task → code region

- Dissimilarity between models
  - $Diss\ (SMM_1, SMM_2) \geq 0$
- Cluster the models
  - Find unusual cluster(s)
  - Use *known* ‘normal’ clustering setting

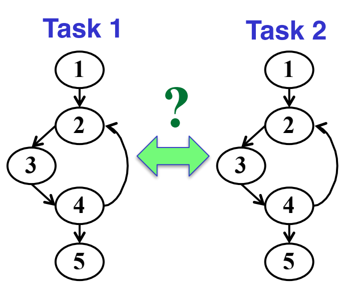


CURRENT WORK

Contributions and Remaining Agenda

- Online fault detection using AutomaDeD
  - Efficient model comparison
  - Scalable faulty-task detection: *CAPEK clustering*, *NN*
- Accurate faulty-task isolation: *model graph compression*
- Evaluation at scale (> 5K processes)

How to Compare Two Semi-Markov Models?

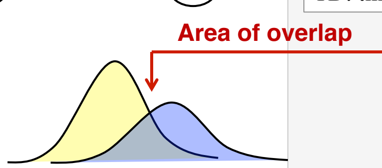


- Compare graph *edge by edge*
- Add up dissimilarities

1 → 2: TP = **0.5**, TD = $N(\mu, \sigma)$

1 → 2: TP = **0.6**, TD = $N(\mu, \sigma)$

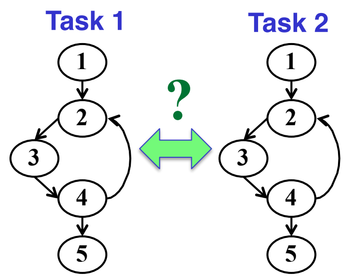
Diss $(\mathrm{SMM}_1, \mathrm{SMM}_2)$
$= \Sigma$ *Lk-norm* (TP) $+ \ \Sigma$ *Lk-norm* (TD),
TP: transition probability
TD: time distribution



Lk-norm

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

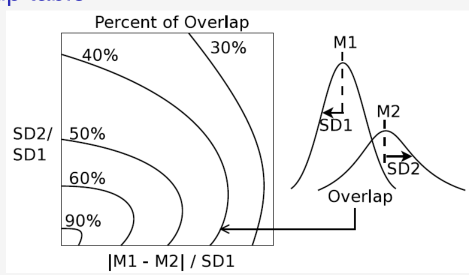
Efficient Comparison of Models' Graphs



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

- Computationally expensive
- Have to evaluate integral for each edge

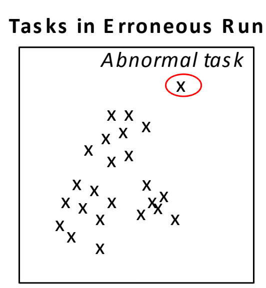
Solution:
Approximate it using pre-computed look-up table



Complexity = constant time

Scalable Faulty-Task Isolation

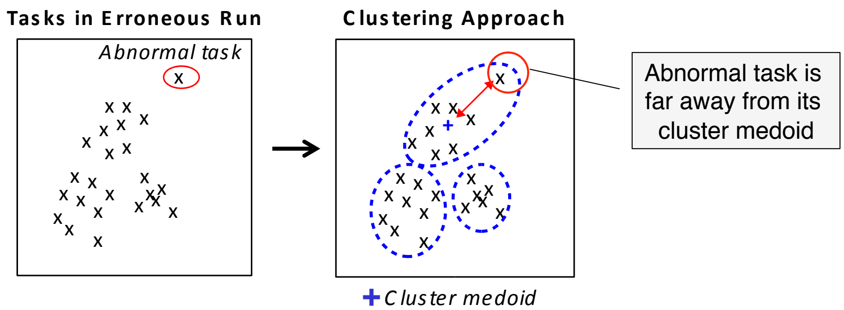
- Typical use-case:
  AutomaDeD isolates abnormal task(s) after failures
  - Input in large-scale applications is often *thousands* of tasks
  - Comparing each task against each other doesn't scale:
    Complexity O(#tasks ^{2})



Challenge:

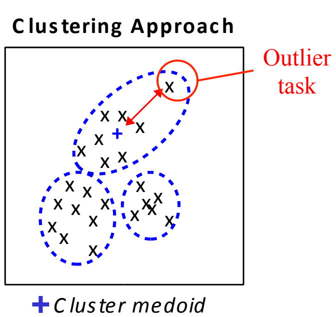
Find scalable algorithm for outlier detection

Faulty-Task Isolation Using Clustering (CAPEK)



- CAPEK, scalable clustering algorithm (ICS '10)
  - Designed for large-scale distributed data sets
  - Based on sampling a constant number of tasks
  - Complexity O(log #tasks)

Algorithm Using Clustering



(1) Perform clustering using CAPEK

(2) Find distances from medoids

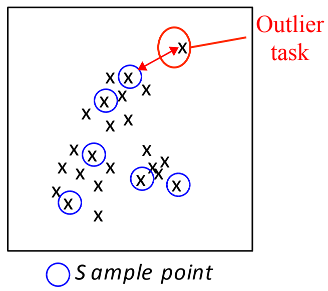
(3) Normalize distances

(4) Find top-k outliers sorting tasks based on the largest distances

- The algorithm is fully distributed
- *Doesn't require a central component to perform the analysis*
- Complexity O(log #tasks)

Faulty-Task Isolation Using Nearest-Neighbor (NN)

Nearest-Neighbor Approach



(1) Sample constant number of tasks

(2) Broadcast samples to all tasks

(3) Find NN distance

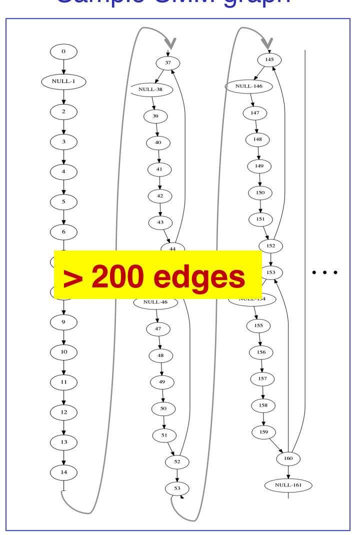
(4) Sort tasks based on distances and select top-k ones

- Assumption is that faulty task will be far from its NN
  - Faster than clustering
  - Works well only when we have one (or a few) faulty task(s)
  - Complexity O(log #tasks)

Lawrence Livermore National Laboratory   Slide 13/23   PURDUE UNIVERSITY

Too Many Graph Edges – *The Curse of Dimensionality*

Sample SMM graph

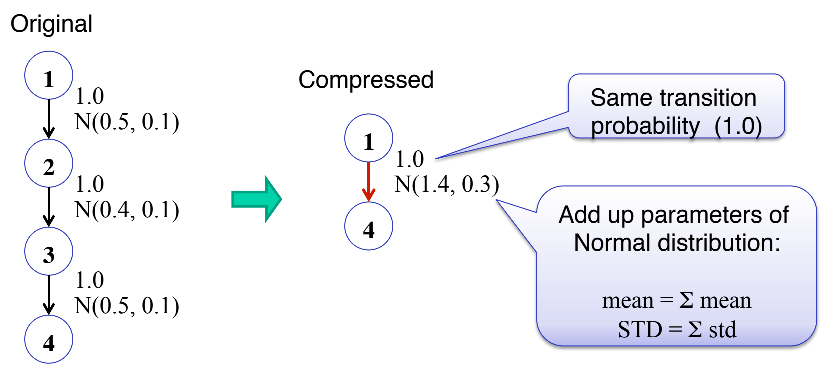


- **Too many edges = Too many dimensions**
- Poor performance of *Clustering* & *Nearest-Neighbor*

Lawrence Livermore National Laboratory   Slide 14/23   PURDUE UNIVERSITY

Graph Compression to Reduce Dimensionality

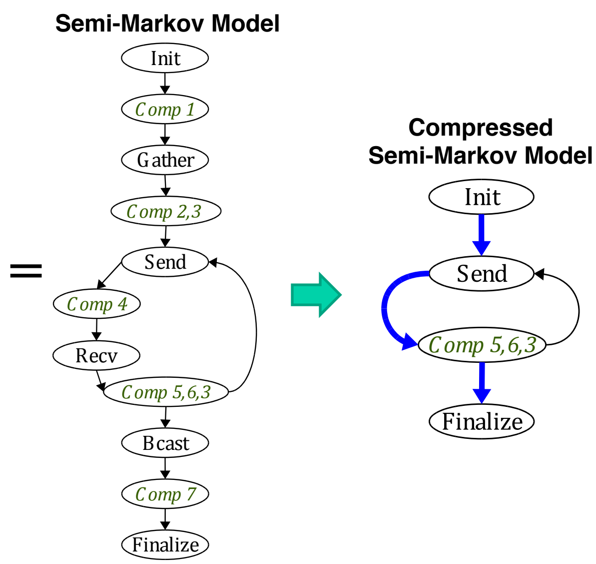
- Sequences of edges can be merged
  - Typically at the beginning / end of program main loop



Example of Graph Compression

Sample code

```
MPI_Init()
//comp code 1
MPI_Gather()
//comp code 2
for (...) {
    //comp code 3
    MPI_Send()
    //comp code 4
    MPI_Recv()
    //comp code 5
}
//comp code 6
MPI_Bcast()
//comp code 7
MPI_Finalize()
```



Fault Injection Types

- We inject faults into the NAS Parallel Benchmarks:
  - BT, SP, CG, FT, LU
  - Injections occur at random {MPI call, task}
  - Linux Sierra cluster at LLNL (six-core nodes, 2.8 GHz, 24GB RAM)
  - Total of 960 experiments

Type	Description
CPU_INTENSIVE	CPU-intensive code region – triply nested loop
MEM_INTENSIVE	Memory-intensive code – filling 1GB buffer at random locations
HANG	Local deadlock – process suspend execution indefinitely
TRANS_STALL	Transient stall – process suspend execution for 5 seconds

Evaluation Metric

- Task Isolation Recall:
  Fraction of runs in which the faulty task (where fault is injected) is in the top-5 abnormal processes

Example:

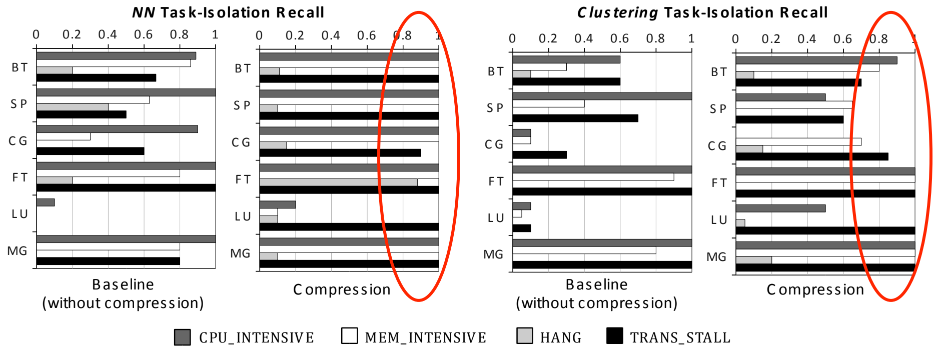
Fault injected in **task 7**

Run 1	Run 2	Run 3
10	25	7
103	158	1
7	3	32
24	1	14
8	10	109
4	103	108
3	24	20
80	73	38

Top-5 abnormal tasks

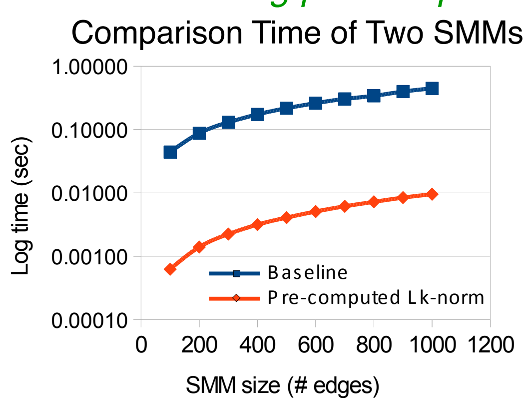
Task-Isolation Recall = 2 / 3 = 0.67

Graph Compression Results



- Compression improves task-isolation recall
  - Dimensionality reduction effectively helps clustering and NN
  - For example, for BT recall of 85% improves to 100% with NN
- NN seems better but injections occur only in one task

Performance Results:
Overhead reduction using pre-computed Lk-norm

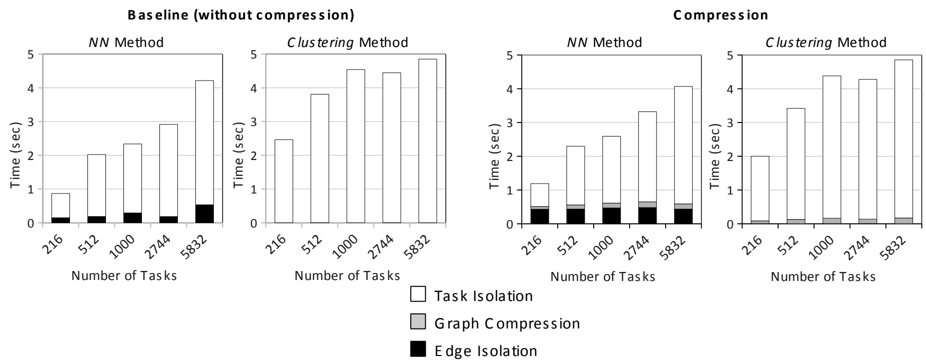


- Using pre-computed table to estimate Lk-norm reduces overhead of comparing SMM graphs
  - Pre-computed method is 46x faster than baseline

Performance Experiments at Scale

- Use Algebraic Multigrid Benchmark (AMG 2006)
  - Scalable multigrid solver
  - Demonstrated up to 125,000 tasks in BlueGene/L
- Ran with over 5,000 tasks in LLNL Sierra Linux cluster
  - Measure time of edge / task isolation and compression

Performance Results:
Time to perform analysis at large scale

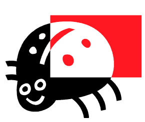


- Compression doesn't incur in much overhead
  - Most of its computation is performed locally
- Entire analysis is performed in less than 5 seconds (5K tasks)
  - Analysis time scales logarithmically w.r.t. number of tasks

Concluding Remarks

- Contributions:
  - Scalable technique to detect faults in MPI applications
  - Implementation scales easily to thousands of tasks
  - Compressing task graph improves anomaly detection accuracy
- Future work:
  - Extend compression technique to allows finer grained instrumentation of function calls
  - Capture more information to detect a wider range of faults

Bring us your fault / bug at large scale



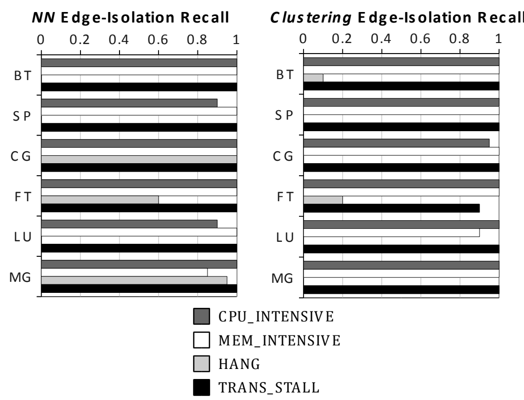
- performance anomaly
- coding bug
…

…we'll be happy to try AutomaDeD on it

Ignacio Laguna <ilaguna@purdue.edu>

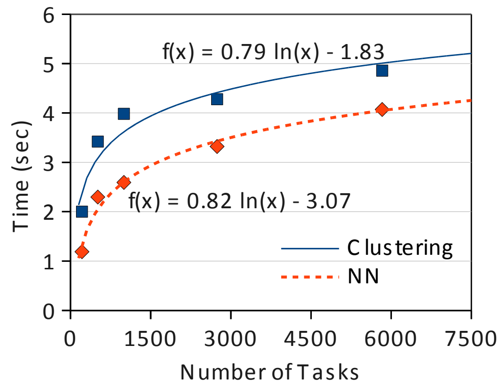
Backup Slides

Edge Isolation Results



- Edge isolation recall is high for both clustering and NN
  - Assumes that faulty task has been correctly identified

Trend Lines for Analysis Time



- Logarithmic curves model the data well
- 8.67 seconds for 10K tasks, 11.29 seconds for 100K tasks