Large Scale Debugging of Parallel Tasks using “Triumph of Majority” Principle and Scaling Properties

Saurabh Bagchi
Dependable Computing Systems Lab (DCSL)
School of ECE, Purdue University

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

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

Greetings come to you from …
A few words about Purdue University

- One of the largest graduate schools in engineering
  - 362 faculty
  - 10,000 students
- Around 40,000 students at its main campus in West Lafayette
- Electrical and Computer Engineering @ Purdue
  - About 900 undergraduate students
  - About 650 graduate students
  - One of the largest producers of Ph.D.s in Electrical and Computer Engineering (about 60 Ph.D's a year)
  - Research expenditure annually around $45M

Debugging Large-Scale Parallel Applications is Challenging

- Large systems will have millions of cores in near future
  - Increased difficulty for developing correct HPC applications
  - Commercial debuggers (gdb, TotalView, DDT) do not perform well at this scale
  - Manual debugging is infeasible
- Faults come from various sources
  - Hardware: soft errors, physical degradation, design bugs
  - Software: coding bugs, misconfigurations
  - A single fault may affect many processes
  - Many instances of silent data corruptions
    - Degrade throughput of clusters
    - Raise doubts about computational answers to serious scientific questions
Developer Steps When Debugging a Parallel Application

Questions a developer has to answer when an application fails:

- Line of code?
- Code region?
- Parallel task that failed?
- When did it fail?

• Need for tools to help developers find root cause quickly

Our Error Detection & Localization Approach

Offline

Phase Annotation

Application

Task_1 \ Task_2 \ \ldots \ Task_n

P_NMPI Profiler

Model_1 \ Model_2 \ \ldots \ Model_n

Offline

Clustering

(1) Abnormal Phases
(2) Abnormal Tasks
(3) Characteristic Transitions
Roadmap

- Problem Motivation
- Application Modeling
  - AutomaDeD’s design
    - Error detection
    - Error localization
- Evaluation of AutomaDeD
- Online detection and localization
- Scale-dependent bugs
- Vrisha’s Design
  - Error detection
  - Error localization
- Evaluation of Vrisha

Types of Behavioral Differences

Run 1
MPI Application Tasks
1 2 3 ... n
 time

Run 2
MPI Application Tasks
1 2 3 ... n
time

Run 3
MPI Application Tasks
1 2 3 ... n
time

Spatial
(between tasks)

Temporal
(between time points)

Between runs
Semi-Markov Models (SMM)

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

![Semi-Markov Model Diagram]

SMM Represents Task Control Flow

• States correspond to:
  – Calls to MPI routines
  – Code between MPI routines

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

foo() {
  MPI_Send(…, 1064, MPI_DOUBLE, …);
  … Computation …
  MPI_Recv(…, 1064, MPI_DOUBLE, …);
  … Computation …
}
```

Application Code

![Semi-Markov Model Diagram for Task Control Flow]
Two Approaches for Time Density Estimation: Parametric and Non-parametric

- **Gaussian Distribution** (Parametric model)
- **Histograms** (Non-parametric model)

**Data Samples**

**Time Values**

- **Bucket Counts**
- **Line Connectors**
- **Gaussian Tail**

- **Density Function**

- **Time Values**

**Comparison**
- **Cheaper**
- **Lower Accuracy**
- **More Expensive**
- **Greater Accuracy**

Roadmap

- **Problem Motivation**
- **Application Modeling**
- **AutomaDeD’s design**
  - Error detection
  - Error localization
- **Evaluation of AutomaDeD**
- **Online detection and localization**
- **Scale-dependent bugs**
- **Vrisha’s Design**
  - Error detection
  - Error localization
- **Evaluation of Vrisha**
### User’s Phase Annotations

**Sample Code:**

```c
main() {
    MPI_Init()
    ... Computation ...
    MPI_Send(…, MPI_INTEGER, …);
    for(…) {
        MPI_Send(…, MPI_DOUBLE, …);
        ...Computation.
        MPI_Recv(…, MPI_DOUBLE, …);
        MPI_Pcontrol();
    }
    ...Computation.
    MPI_Recv(…, MPI_INTEGER, …);
    MPI_Finalize();
}
```

- Phases denote *regions of execution* repeated dynamically
- Developers annotate phases in the code
  - *MPI_Pcontrol* is intercepted by wrapper library
A Semi-Markov Model per Task, per Phase

Task 1
SMM
SMM
SMM
SMM

Task 2
SMM
SMM
SMM
SMM

Task n
SMM
SMM
SMM
SMM

Phase 1
SMM
SMM
SMM
SMM

Phase 2
SMM
SMM
SMM
SMM

Phase 3
SMM
SMM
SMM
SMM

Phase 4
SMM
SMM
SMM
SMM

time

time

time

time

AutomaDeD’s Error Detection Approach

Offline
Phase Annotation

Application
Task_1 Task_2 ... Task_n

Online
PNNMPI Profiler
Model_1 Model_2 ... Model_n

Offline
Clustering

(1) Abnormal Phases
(2) Abnormal Tasks
(3) Characteristic Transitions
Faulty Phase Detection: Find the Time Period of Abnormal Behavior

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

**Sample runs available:**

Without sample runs:

Deviation score

Comparing each phase to all others

**Clustering Tasks’ Models:** Hierarchical Agglomerative Clustering (HAC)

\[
\text{Diss}(\text{SMM}_1, \text{SMM}_2) = L_2 \text{ Norm (Transition prob.)} + L_2 \text{ Norm (Time prob.)}
\]

- **Step 1:** Each task starts in its own cluster
  - Task 1 SMM
  - Task 2 SMM
  - Task 3 SMM
  - Task 4 SMM

- **Step 2:**
  - Task 1 SMM
  - Task 2 SMM
  - Task 3 SMM
  - Task 4 SMM

- **Step 3:**
  - Task 1 SMM
  - Task 2 SMM
  - Task 3 SMM
  - Task 4 SMM

- **Step 4:**
  - Question mark
  - Do we stop? or,
  - Do we get one cluster?

We need a dissimilarity threshold to decide when to stop
How To Select The Number Of Clusters

- User provides application’s natural cluster count $k$

- Use sample runs to compute clustering threshold $\tau$ that produces $k$ clusters
  - Use sample runs if available
  - Otherwise, compute $\tau$ from start of execution
  - Threshold based on highest increase in dissimilarity

- During real runs, cluster tasks using threshold $\tau$

Cluster Isolation Example

Cluster Isolation: to separate buggy task in unusual cluster

Master-Worker Application Example

Normal Execution

Cluster 1
Cluster 2

Buggy Execution

Cluster 1
Cluster 2
Cluster 3

Bug in Task 9
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)

Roadmap

- Problem Motivation
- Application Modeling
- AutomaDeD’s design
- **Evaluation of AutomaDeD**
  - Synthetic injections
  - Real-world bug
- Online detection and localization
- Scale-dependent bugs
- Vrisha’s Design
  - Error detection
  - Error localization
- Evaluation of Vrisha
Fault Injections

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

- 2000 injection experiments per application:

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

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
Cluster Isolation Accuracy: Isolating the abnormal task(s)

- Results assume phase detected accurately
- Accuracy of Cluster Isolation highly variable

Accuracy up to 90% for extra threads

Poor detection elsewhere because of fault propagation: buggy task → normal task(s)

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
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

MVAPICH Bug

- Job execution script failed to clean up at job end
  - MPI tasks executer (mpirun, version 0.9.9)
  - Left runaway processes on nodes

- Simulation:
  - Execute BT (affected application)
  - Run concurrently runaway applications (LU, MG or SP)
  - Runaway tasks interfere with normal BT execution
MVAPICH Bug Results:

**SMMs Deviation Scores in Affected Application**

Affected application: BT benchmark
Interfering applications: SP, LU, MG benchmarks

<table>
<thead>
<tr>
<th>Phase</th>
<th>16-task BT / 16-task SP/LU/MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>1E+5</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>1E+4</strong></td>
</tr>
<tr>
<td>3</td>
<td><strong>1E+3</strong></td>
</tr>
<tr>
<td>4</td>
<td><strong>1E+2</strong></td>
</tr>
<tr>
<td>5</td>
<td><strong>1E+1</strong></td>
</tr>
<tr>
<td>6</td>
<td><strong>1E+0</strong></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Abnormal phase detected in phase 1 in SP and LU, and in phase 2 in MG
Constant (average) SMM difference in regular BT runs

---

Roadmap

- Problem Motivation
- Application Modeling
- AutomaDeD’s design
- Evaluation of AutomaDeD
- Online detection and localization
  - Scalability challenges
  - Scalability solutions
  - Performance evaluation
- Scale-dependent bugs
- Vrisha’s Design
  - Error detection
  - Error localization
- Evaluation of Vrisha
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 8.67 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
    \[ \int_{-\infty}^{\infty} |P(x) - Q(x)|^\delta \, dx \]
  - 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 space is hard.
  - Complexity of difference computation proportional to number of edges.

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.
Performance Result with Scalability Techniques

- Application: Algebraic MultiGrid (AMG) 2006 benchmark
- Cluster has nodes, each of which has 6 cores with 2.8 GHz Intel processors and 64 GB of 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

Roadmap

- Problem Motivation
- Application Modeling
- AutomaDeD’s design
- Evaluation of AutomaDeD
  - Synthetic injections
  - Real-world bug
- Online detection and localization
- Scale-dependent bugs
- Vrisha’s Design
  - Error detection
  - Error localization
- Evaluation of Vrisha
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

When Statistical Debugging Meets Scale-dependent Bugs

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

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
Contributions

- We build **Vrisha** to use scale of run as a parameter to 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

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
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

```c
int MPIR_Allgather (  
  int recvcount,  
  MPI_Datatype recvtype,  
  MPID_Comm *comm_ptr )  
{
  int comm_size, rank;  
  int curr_cnt, dst, type_size, left, right, jnext, comm_size_is_pof2;  
  ....

  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-  
            * no. of processes, use ring algorithm. */
    ....
  }
```
Roadmap

- Problem Motivation
- Application Modeling
- AutomaDeD’s design
- Evaluation of AutomaDeD
  - Synthetic injections
  - Real-world bug
- Online detection and localization
- Scale-dependent bugs
- Vrisha’s Design
  - Error detection
  - Error localization
- Evaluation of Vrisha

Key Observation

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

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

Large gap means bug detected!

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
Challenges

• What features should we use to build the model of scaling behavior?

Control Feature

• 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
Observational Feature

- To characterize program behavior, we use **observational features**
- A set of vantage points in source code to profile various runtime properties
- Observational features are the manifestations of program behavior

Observational Feature

- What does Vrisha use for observational feature?
  - Each unique call stack at the socket level under the MPI library as a distinct vantage point
  - Record the amount of data communicated at each vantage point as observational feature
- Why?
  - Capture both application and library bugs
  - Not need to modify application code
  - Impose low overhead
  - Provide source level diagnosis
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

\[
\max_{u, v} \ \text{corr}(X_u, Y_v)
\]

such that \( \|u\| = \|v\| = 1 \)
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 $\varphi(.)$.

Challenges

- How does Vrisha do bug detection and diagnosis?
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.

Bug Localization

- Normalize communication volume of training and production runs to the same scale.
- Comparing training (bug-free) with production (buggy) runs.
- Identify the call stacks for the biggest changes as the potential location for bug.
Roadmap

- Problem Motivation
- Application Modeling
- AutomaDeD’s design
- Evaluation of AutomaDeD
  - Synthetic injections
  - Real-world bug
- Online detection and localization
- Scale-dependent bugs
- Vrisha’s Design
  - Error detection
  - Error localization
- Evaluation of Vrisha

Evaluation

- We validated Vrisha with two real bugs from the MPICH2 library and Vrisha was able to detect and localize both of them
- Experiment setup
  - 16-node cluster
  - dual-socket 2.2GHz quad-core CPU
  - 512K L2, 8G RAM
  - MPICH2-1.1.1
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

Locate the Bug in Allgather

- All the training runs like 4- and 8-node runs share the same pattern
- The 16-node run shows a different pattern
- The most radical difference happens between feature #9 and #16 which leads us to find the program makes a wrong choice at the buggy if statement
Vrisha’s Overhead

- We measured the average overhead of Vrisha with NAS Parallel Benchmarks
- Vrisha’s instrumentation overhead is less than 8% execution time on average
- Vrisha takes less than 30 ms to build model and less than 5 ms to detect bug, equivalent to around 1/10000 of the running time of these benchmarks on average

Concluding Remarks

- 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
  - Accuracy up to ~90% for phase detection, cluster and transition isolation (delays and hangs)
  - Scalable technique
  - For a subtle class of bugs called scale-dependent bugs, a technique that estimates scale-determined properties
- Insights:
  - Different machine learning techniques should part of our toolkit
  - Different kinds of bugs need different approaches
  - Detection and localization significantly aided by training runs and developer hints