Probabilistic Diagnosis of Performance Faults in Large-Scale Parallel Applications

Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, Todd Gamblin

Developing Resilient HPC Applications is Challenging

MTTF of hours in Future Exascale Supercomputers

Faults come from:
- Hardware
- Software
- Network

Software bugs from:
- Application
- Libraries
- OS & Runtime system

Multiple manifestations:
- Hangs, crashes
- Silent data corruption
- Applications is slower than usual
Some Faults Manifest Only at Large Scale

Molecular dynamics simulation code (ddcMD)

Fault Characteristics

- Application hangs with 8,000 MPI tasks
- Only fails in Blue Gene/L
- Manifestation was intermittent
- Large amount of time spent on fixing the problem
- Out technique isolated the origin of the problem in a few seconds

Why Do We Need New Debugging Tools?

Current Tools

- Old (breakpoint) technology (>30 years old)
- Manual process to find bugs
- Poor scalability

Future Tools

- Automatic problem determination
- Less human intervention in determining failure root cause
- Scalable (~millions of processes)
Approach Overview

- We focus on pinpointing the origin of performance faults:
  - Application hangs
  - Execution is slower than usual
- Could have multiple causes:
  - Deadlocks, slower code regions, communication problems, etc.

1. Model parallel tasks behavior
2. Find faulty task(s)
3. Find problematic code region

Summarizing Execution History

- HPC applications generate a large amount of traces
- Use a probabilistic model to summarize traces
- We model control flow behavior of MPI tasks
  - Allow us to find the least progressed task
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

- **MPI_Gather**
  - **Comp. Code 1**
  - **Comp. Code 2**
  - **Comp. Code 3**

- **MPI_Send**
  - **Comp. Code 1**
  - **0.75**

- **MPI_Recv**
  - **0.3**
  - **0.6**

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

Approach Overview

1. Model parallel tasks behavior
2. Find faulty task(s)
3. Find problematic code region
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*

What Tasks are Progress Dependent On Other Tasks?

**Point-to-Point Operations**

Task X:

```c
// 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:

```c
// 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)
Probabilistic Inference of Progress-Dependence Graph

Sample Markov Model

Task A

1.0

0.7

Task B

3

0.3

2

1.0

Task C

5

1.0

0.9

6

0.1

... Task D

8

9

1.0

1.0

Task E

7

10

1.0

1.0

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

Resolving Conflicting Probability Values

Sample Markov Model

Task A

1.0

0.7

Task B

3

0.3

2

1.0

Task C

5

1.0

0.9

6

0.1

... Task D

8

9

1.0

1.0

Task E

7

10

1.0

1.0

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
Distributed Algorithm to Infer the Graph

Tasks
1 x 2 y 3 z ... n

Time
All-reduction of current states

All tasks know the state of others

Build (locally) progress-dependence graph

Reduction of progress-dependence graphs

Reductions are $O(\log \#\text{tasks})$

Progress dependence graph

Reduction Operations: Graph Dependencies Unions

Examples of reduction operations
$X \rightarrow Y$: $X$ is progress dependent on $Y$

<table>
<thead>
<tr>
<th>Task A</th>
<th>Task B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \rightarrow Y$</td>
<td>$X \rightarrow Y$</td>
<td>$X \rightarrow Y$ (Same dependence)</td>
</tr>
<tr>
<td>$X \rightarrow Y$</td>
<td>Null</td>
<td>$X \rightarrow Y$ (First dominates)</td>
</tr>
<tr>
<td>$X \rightarrow Y$</td>
<td>$Y \rightarrow X$</td>
<td>Undefined (or Null)</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>
Bug Progress Dependence Graph

Hang with ~8,000 MPI tasks in BlueGene/L

[3136] Least-progressed task

[0, 2048, 3072]

[0-2047, 3073-3135, ...]

[6840]

[6841-7995]

- Our tool finds that task 3136 is the origin of the hang
- How did it reach its current state?

Approach Overview

1. Model parallel tasks behavior
2. Find faulty task(s)
3. Find problematic code region
Finding the Faulty Code Region: *Program Slicing*

```
done = 1;
for (...) {
    if (event) {
        flag = 1;
    }
}
if (flag == 1) {
    MPI_Recv();
    ...
}
if (done == 1) {
    MPI_Barrier();
}
```

**DynInst**
Slicing Tool

---

**Case Study**

**Code to Handle Buffered I/O in DDCMD**

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

**Writer:**
- `Send()`
- `Recv()`

**Non-Writer**
- `Recv()`
- `Send()`

**Signals**

**Data**

Check if another writer asks for data
Slice With Origin of the Bug

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

Dual condition occurs in BlueGene/L
• A task is a writer and a non-writer

MPI_Probe checks for source, tag and comm of a message
• Another writer intercepted wrong message

Programmer used unique MPI tags to isolate different I/O groups
```

Least-progressed task State

Fault Injections Methodology

• Faults injected in Two Sequoia benchmarks:
  – AMG-2006
  – LAMMPS

• We injected a hang in random MPI tasks:
  – 20 user function calls, 5 MPI calls
  – Only injected in executed functions
  – Functions are selected randomly
Accurate Detection of Least-Prog. Tasks

- Least-progressed task detection recall:
  - Cases when LP task is detected correctly
- Imprecision:
  - % of extra tasks in LP tasks set

**Example Runs:**
- 64 tasks, fault injected in task 3

- Example 1
  - LP task detected (Imprecision = 0)

- Example 2
  - LP task detected (Imprecision = 2/3)

- Overall results:
  - Average LP task detection recall is 88%
  - 86% of injections has imprecision of zero

Performance Results:

*Least-Prog. Task Detection Takes a Fraction Of A Second*
Performance Results:

**Slowdown Is Small For a Variety of Benchmarks**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slowdown</th>
<th>Memory-usage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMMPS</td>
<td>1.59</td>
<td>6.11</td>
</tr>
<tr>
<td>AMG2006</td>
<td>1.46</td>
<td>10.36</td>
</tr>
<tr>
<td>BT</td>
<td>1.08</td>
<td>3.75</td>
</tr>
<tr>
<td>SP</td>
<td>1.67</td>
<td>5.14</td>
</tr>
<tr>
<td>CG</td>
<td>1.14</td>
<td>2.21</td>
</tr>
<tr>
<td>FT</td>
<td>1.05</td>
<td>1.01</td>
</tr>
<tr>
<td>LU</td>
<td>1.39</td>
<td>5.37</td>
</tr>
<tr>
<td>MG</td>
<td>1.04</td>
<td>1.04</td>
</tr>
</tbody>
</table>

- Tested slowdown with NAS Parallel and Sequoia benchmarks
  - Maximum slowdown of ~1.67
- Slowdown depends on number of MPI calls from different contexts

**Conclusions**

- Our debugging approach diagnose faults in HPC applications

**Novelties:**
  - Compression of historic control-flow behavior
  - Probabilistic inference of the least-progressed tasks
  - Guided application of program slicing

- Distributed debugging method is scalable
  - Takes fraction of a second with 32,000 BlueGene/P tasks

- Successful evaluation with hard-to-detect bug and representative benchmarks
Thank you!

Backup Slides
Dual Role Due to BlueGene/L I/O Structure

Node X

Task 1
Task 2

Group A

Node Y

Task 5
Task 6

Group B

Dual Role
Task 6:
Non-writer for its own group (B)
Writer for a different group (A)

• In BlueGene/L, I/O is performed through dedicated nodes
• Program nominates only one task per I/O node