Noise Injection Techniques to Expose Subtle and Unintended Message Races

PPoPP2017
February 6th, 2017

Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, Martin Schulz and Christopher M. Chambreau
Debugging large-scale applications is challenging

“On average, software developers spend 50% of their programming time finding and fixing bugs.”[1]


In HPC, applications run in parallel which makes debugging particularly challenging
“MPI non-determinism” makes debugging applications even more complicated

- MPI supports wildcard receives
  - MPI processes can wait messages from any MPI processes

- Message receive orders can change across executions
  - Due to non-deterministic system noise (e.g. Network, OS jitter)

➡️ MPI non-deterministic application which correctly ran in first execution can crash in the second execution even with the same input
Real-world non-deterministic bugs in Diablo/Hypre 2.10.1*

- MPI non-deterministic bugs cost computational scientists substantial amounts of time and efforts

Diablo/Hypre 2.10.1

The scientists
- It hung only once every 50 runs after a few hours
- The scientists spent **2 months** in the period of **18 months**, and then gave up on debugging it

Our debugging team
- We found that the cause is due to a "Unintended message matching" by misused MPI tag (message race bug)
- We spent **2 weeks** in the period of **3 months** to fix the bug

* Hypre is an MPI-based library for solving large, sparse linear systems of equations on massively parallel computers
Observing a non-deterministic bug is costly

- Due to such non-determinism, we needed to submit a bunch of debug jobs to observe the bug
  - The bug did not manifest in 98% of jobs
  - Wasted 9,560 node-hour

- Rarely-occurring message race bugs waste both scientists' productivity and machine resources (thereby affect also other users)

A tool to frequently and quickly expose message race bugs is invaluable
NINJA

- **NINJA: Noise Injection Agent**
  - Frequent manifestation: Injects network noise in order to frequently and quickly expose message race bugs
  - High portability: NINJA is developed in MPI profiling layer (PMPI)

- **Experimental results**
  - NINJA consistently manifests the Hypre 2.10.1 message race bug which does not manifest itself without NINJA
Outline

- Introduction
- Message race bugs
- NINJA: Noise Injection Agent
- Evaluation
- Conclusion
Data-parallel model (or SPMD)

- In HPC, many applications are written based on a data-parallel model (or SPMD)
  - Easy to scale out the application by simply dividing a problem across processes
- In SPMD, each process calls the same series of routines in the same order
- So messages sent in a communication routine are all received within the same communication routine
  ➞ “self-contained” communication routine (or communication routine)
Plots of Send and Receive time stamps

- HPC apps call a series of self-contained communication routines step-by-step
  - Each colored box illustrates a self-contained routine
Avoiding message races

- To make communication routines “self-contained”, common approaches in MPI are:
  - Use of different tags/communicators
  - Calling synchronization (e.g. MPI_Barrier)

If these conditions are violated, applications potentially embrace message race bugs.
Message race bugs are non-deterministic

- Manifestations of message race bugs depend on system noise
  - Occurrences and amounts of system noise are non-deterministic
- Message race bugs rarely manifest, E.g., when
  1. System noise level is low
  2. Unsafe routines (Routine A and Routine B) are separated by interleaving routines (Routine X)

Correct message matching

Wrong message matching

Routine A

Routine X

Routine B

P0  P1  P2

Noise

Crash
Case study: Diablo/Hypre 2.10.1

- The message race bug in Hypre manifest when a message sent in Routine 3 is received in Routine 1
  - Routine 1 & 3: same MPI tag without synchronization

However, Routine 1 and 3 are significantly separated by 2.5 msec, the message race bug rarely manifest.

We need a tool to frequently expose subtle message race bugs.
NINJA: Noise Injection Agent Tool

- NINJA emulates noisy environments to expose subtle message race bugs

  Correct message matching

  Wrong message matching

- Two noise injection modes
  - System-centric mode: NINJA emulates congested network to induce message races
  - Application-centric mode: NINJA analyzes application’s communication pattern, and inject a sufficient amount of noise to make two unsafe routines overlapped
System-centric mode emulates noisy network

- System-centric mode emulates noisy network based on a conventional “flow control” in interconnects

- Conventional flow control
  - When sending a message, the message is divided into packets and queued into a send buffer
  - The packets are transmitted from a send buffer to a receive buffer
  - If the receive buffer does not have enough space, flow control engine suspends packet transmission until enough buffer space is freed up
NINJA implements flow control at process-level

- NINJA’s flow control
  - Each process manages virtual buffer queue (VBQ)
  - If VBQ does not have enough space, NINJA delays sending the MPI message until enough buffer space is freed up
How NINJA triggers noise injection?

- NINJA system-centric mode
  - Monitor # of incoming packets
  - Compute # of outgoing packets by using a model based on network bandwidth and latency
  - Estimate VBQ length
  - If VBQ length exceeds the VBQ size, then NINJA injects noise to the message

- NINJA logically estimate VBQ length, so does not physically buffer messages by copying

![Diagram of NINJA noise injection process]

\[ \text{MPI processes} \]

\[ \text{NIC} \]

\[ \text{Physical link} \]

\[ \sum_{i=1}^{n} \left( \frac{P_s[i]}{B + C} \right) \]

\[ \text{# of incoming packets} \]

\[ \text{# of outgoing packets} \]

\[ \text{VBQ size} \]

\[ \text{VBQ length} \]
How much amount of noise is injected?

- NINJA delay a message send until enough VBQ space is freed up

Example
- VBQ size: 5 packets
- # of packets in VBQ: 3 packets
- The incoming message: 4 packets
  ➔ NINJA delays this message by the time to transmit 2 packets

\[
\text{Packet size} = 2 \text{ [KB]}
\]
\[
B = 3.14 \text{ [GB/sec]}
\]
\[
C = 0.25 \text{ [µsec]}
\]

\[
\left( \frac{2 \text{ [KB]}}{3.14 \text{ [GB/sec]}} + 0.25 \text{ [µsec]} \right) \times 2 \text{ packets} = 1.27 \text{ [msec]}
\]

Noise amount
System-centric mode induces message races

- Earlier messages are not delayed in a routine (since buffer space is left) while later messages are delayed in the same routine.
- NINJA extends an unsafe routine so that we can overlap one unsafe communication routine with the next communication routine, thereby, induce message races.
Application-centric mode

- Problem in system-centric mode
  - If unsafe routines (i.e. Routine A and B) are significantly separated, system-centric noise amount is not adequate

- Application-centric mode
  - NINJA analyzes communication patterns during system-centric mode
  - Then, NINJA injects an adequate amount of noise to enforce message races
Application-centric mode

1. Each process traces message send time stamps

![Diagram showing message send call and send interval in system-centric mode. The diagram includes a timeline from 0 to 300 with marked time points and arrows indicating message send calls.](image-url)
Application-centric mode

2. Compute message send intervals based on the time stamps
Application-centric mode

3. Detect separated unsafe routines
   — If an interval is more than system-centric noise amount, NINJA regards the routines as separated unsafe routines
   — Example
     • System-centric noise amount: 20 μsec
     • NINJA regards Set 1 and 2 as separated unsafe routines more than system-centric noise amount

![Diagram showing time and message send calls with set 1 and set 2 highlighted, and a 20 μsec interval marked.]
Application-centric mode

4. Compute this separated interval between the two routines

- Sum of intervals:

\[
\sum_{k=m_i}^{m_{i+1}-1} D_k
\]

- Updates max of this separated interval every iterations for every detected pairs of separated routines
Application-centric mode

- At the end of system-centric mode, each process writes this analysis file
- Application-centric mode read this file and inject noise according this analysis
  - i.e. System-centric mode with auto-tuned noise amount

Execution in application-centric mode

180 [μsec]

Execution in system-centric mode

- <tag1, comm1> 180 [μsec]
- <tag2, comm1> 65 [μsec]
- <tag2, comm2> 230 [μsec]
- <tag4, comm2> 1500 [μsec]
Implementation

- We implement the noise injection schemes by using PMPI profiling interface
- To inject network noise, we use a send-dedicated thread, one per MPI process
  - (1) MPI_Init,
    - Each MPI process spawns this send-dedicated thread
  - (2) MPI_Isend for non-delayed messages
    - Calls PMPI_Isend
  - (3) MPI_Isend for delayed messages
    - The main thread calls PMPI_Send_init, computes the amount of delay, and set delayed send time
  - (4) PMPI_Start
    - The send thread periodically check the send time
    - When the scheduled send time comes, the send thread calls PMPI_Start
Evaluation

- **Cases**
  - Two synthetic benchmarks: Case 1 and 2
  - Parasail module in Hypre 2.10.1
    - Computes a sparse approximate inverse pre-conditioner, which is used by Diablo

- **Environment**
  - MVAPICH-2.1
  - LLNL systems
    - Run 64 processes in 4 nodes

<table>
<thead>
<tr>
<th></th>
<th>Cab</th>
<th>Catalyst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>1,200 batch nodes</td>
<td>304 batch nodes</td>
</tr>
<tr>
<td>CPU</td>
<td>2.6 GHz Intel Xeon E5-2670 (16 cores per node)</td>
<td>2.4 GHz Intel Xeon E5-2695 v2 (24 cores per node)</td>
</tr>
<tr>
<td>Memory</td>
<td>32 GB</td>
<td>128 GB</td>
</tr>
<tr>
<td>HCA</td>
<td>InfiniBand QDR4X (QLogic)</td>
<td>InfiniBand QDR4X (QLogic) x2</td>
</tr>
</tbody>
</table>

- **Evaluate the number of loops at which a message race occurs**
Case 1: Send-Receive

1. In Cab, this message race easily manifest itself without NINJA because Cab is relatively noisy system

2. In less noise system, this message race rarely manifest

3. If we use NINJA in catalyst, we can frequently and immediately manifest message race even in this less noisy system
Case 2: Send-AllReduce-Receive

Typical communication patterns when each MPI rank does not know how many messages arrive

- **P0**: Send to P1 and P2
- **P1**: Send to P0
- **P2**: Send to P0 and P1
Case 2: Send-Allreduce-Receive with 1 msec interval

1. Message race does not manifest at all even in Cab

2. System-centric noise also cannot manifest the message races because noise amount is too small for these unsafe routine separated by 1 [msec]

Max Iterations: 1,000

3. Application-centric noise can consistently and immediately manifest message races because this mode analyzes how much unsafe routines are separated and injects adequate amount of noise
Hypre 2.10.1

- NINJA also successfully manifest real message race bugs with application-centric mode

Unsafe communication routines in Hypre 2.10.1
Discussion

- Disadvantage: NINJA cannot reproduce the same message race → However, the same message race can be reproduced by using MPI record-and-replay technique

Conclusion

- Debugging large-scale HPC applications are becoming more challenging

- Rarely-occurring message race bugs hamper debugging productivity because they do not frequently manifest

- NINJA can frequently and immediately manifest such message race bugs

- As future work, we will integrate NINJA with ReMPI
  - Currently, NINJA and ReMPI are independent tools
Thanks!

**Git repository:**

NINJA: PRUNER NINJA OR https://github.com/PRUNERS/NINJA

ReMPI: PRUNER ReMPI OR https://github.com/PRUNERS/ReMPI

**Speaker:**

Kento Sato (佐藤 賢斗)
Lawrence Livermore National Laboratory
https://kento.github.io

**Team members**

Dong H. Ahn, Ignacio Laguna, Gregory L. Lee,
Martin Schulz and Chambreau, Chris

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-PRES-720797).