Noise Injection Techniques to Expose Subtle and Unintended Message Races

Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee,
Martin Schulz and Christopher M. Chambreau

Lawrence Livermore National Laboratory
7000 East Ave. Livermore, CA
{kento, ahn1, ilaguna, lee218, schulzm, chambreau1}@llnl.gov

Abstract

Debugging intermittently occurring bugs within MPI applications is challenging, and message races, a condition in which two or more sends race to match with a receive, are one of the common root causes. Many debugging tools have been proposed to help programmers resolve them, but their runtime interference perturbs the timing such that subtle races often cannot be reproduced with debugging tools. We present novel noise injection techniques to expose message races even under a tool's control. We first formalize this race problem in the context of non-deterministic parallel applications and use this analysis to determine an effective noise-injection strategy to uncover them. We codified these techniques in NINJA (Noise INJection Agent) that exposes these races without modification to the application. Our evaluations on synthetic cases as well as a real-world bug in Hypre-2.10.1 show that NINJA significantly helps expose races.

Categories and Subject Descriptors D.2.5 [*Software*]: Distributed debugging; F.1.2 [*Theory of Computation*]: Alternation and nondeterminism

Keywords debugging; non-determinism; MPI

1. Introduction

The path towards exascale computing involves significant increases in the amount and type of parallelism, and applications have begun to adopt more asynchronous algorithms to exploit them efficiently. The Message Passing Interface (MPI) is today's workhorse to build large-scale high-performance computing (HPC) applications. While this programming model will remain dominant in the foreseeable future, its usage is increasingly embracing more asynchronous algorithms.

While asynchronous communication allows flexible communication patterns and enables applications to overlap communication with computation [20] for higher performance, it comes at an expense: it often introduces *non-determinism* in communication patterns, e.g., by waiting for any message (e.g. `MPI_ANY_SOURCE`) to arrive and processing them in any order. While this by itself is not problematic, non-determinism is generally hard to reason about, raising the level of difficulty in correctly implementing such algorithms. Even worse, incorrect usage can often produce subtle and hard-to-track non-deterministic bugs that intermittently emerge during production runs, but rarely (or never) occur in testing and debug runs.

A common cause of a non-deterministic bug is an unintended message race, a condition in which two or more message sends race to match with a message receive and at least one of them is unintended. Such an incorrect match only occurs with under a specific message arrival timing, and therefore they can be particularly challenging to reproduce and to diagnose during debug runs, which typically have a different timing behavior caused by different compilation flags, additional instrumentation or other factors. Further, timing is often determined also by external factors in MPI applications, in particular the level of congestion on the network paths.

Unintended message races can significantly increase the debugging cost because programmers must often run the application many times until the races finally manifest. While many parallel debugging tools exist for HPC, they mainly focus on helping programmers in *finding* the cause of a bug by inspecting, tracing, profiling or printing the application's program state once a bug has manifested. Consequently, these tools are effective *only when the bug is observed*. However, *observing* the bug itself, especially under a tool's control, is as difficult as *finding* the root cause of the bug for unintended message races (More details in Sec-

PPoPP '17, Feb. 4–8, 2017, Austin, Texas, USA..
Copyright © 2017 ACM 978-1-4503-4493-7/17/02…$15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3018743.3018767

tion 2). Further, these tools typically exhibit noticeable runtime overhead, which distorts and thereby potentially masks message race bugs.

In this paper, we present NINJA (Network noise INJection Agent), which injects network noise in order to expose unintended message races more frequently and quickly to *complement* existing debugging tools. The noise amounts are carefully selected to cause delays only on certain message sends to expose the condition akin to when the application intermittently failed.

NINJA provides two network injection modes: system-centric and application-centric. In the system-centric mode, NINJA is configured with system-specific noise parameters to emulate a congested network with no knowledge on the application's messaging patterns. In the application-centric mode, NINJA first obtains the application's own messaging patterns from an initial run in the system-centric mode and then uses them to determine more precise amounts of noise and injection patterns. NINJA uses the MPI profiling interface (PMPI) to inject network noise, and thus existing applications can run under NINJA without code modifications or even without being aware of the tool's existence. Specifically, this paper makes the following contributions:

- An analysis of unintended message races in non-deterministic message-passing applications;
- Two novel network noise injection techniques: one system- and application-centric approach, which use the above analysis to uncover subtle message races;
- A prototype of NINJA that embodies these techniques;
- An empirical evaluation of the effectiveness and overhead of NINJA on synthetic message races as well as a real-world bug in a large sparse matrix solver.

Our evaluation on synthetic cases shows that NINJA can help expose unintended message races frequently and quickly. Further, our evaluation on a real-world bug in Hypre-2.10.1 shows that NINJA can consistently expose message races that had not been fixed (due in large part to their rare occurrence) for over 14 years of its history, i.e., since the first version of Hypre 1.6.0 was released[1].

2. Background

2.1 Motivational Case Study

Two scientists developing a production engineering application in our organization engaged our research team to ask for debugging assistance as their application intermittently hung. They had spent about two months on the hangs in a period of 18 months, but its intermittent nature had been making their effort nearly fruitless. Our research team began to tackle this problem by first lightly sprinkling `printf` statements into code and running it on one of our HPC clusters. However, we found that the bug became even harder to reproduce in our debug runs (more details in Section 3.3).

When our team found and fixed the bugs, our job submission logs showed that our team had to spend two weeks in the period of three months, and submitted 291 jobs (10,392 compute-node hours or 433 compute-node days in total) for diagnosing this single non-deterministic bug. We were able to observe the hang only in 9 jobs, and wasted resources for the remaining 282 jobs. Even when the hang was reproduced in the 9 jobs, it took a few hours before the application got hung. Another phenomena, which hampered our debugging efforts, was that this bug never manifested in another cluster (More details in Section 6.1).

Once we sufficiently captured the hang in debug runs, fixing it was rather easy: it required only about ten lines of code. The root cause was due to an unintended message race between two communication routines within the Hypre solver (Ver. 2.10.1) used in this application. We learned in this investigation that debugging non-deterministic bugs requires frequent and quick manifestations of the target bug.

2.2 Existing Approaches to Message Races

Because of its non-deterministic nature, `printf` debugging is a common technique: programmers add `printf` statements in specific places of the code to log program states including destinations and sources of the messages being exchanged.

Other attractive approaches are record-and-replay [16, 20, 26], MPI deadlock detection [12] and message race detection tools [19]. A record-and-replay tool records a communication trace of all MPI processes. Once the tool captures an incorrect message matching in the record phase, the tool can then repeatedly reproduce the identical incorrect execution, significantly facilitating debugging. However a record-and-replay tool needs to observe untended message matching at lease once and under tool control in order to replay it.

MPI deadlock and message race detection tools can identify where a deadlock or message race occurred in the code. However, these tools also need to observe a deadlock or message race to identify the code location. When it comes to non-deterministic bugs, a new tool, which can make the target bug manifest itself more frequently and quickly so as to complement other existing debugging tools, can be a significant aid.

3. Conceptualizing Message Races

In this section, we present our high-level execution model for MPI applications in order to describe message races in conceptual terms. Using this model, we also explain a common phenomenon where simply applying a debugging tool to this problem often makes it even harder to expose this class of bugs.

[1] The corrected functions, which our research team found and fixed, were incorporated into the next version, Hypre-2.11.0.

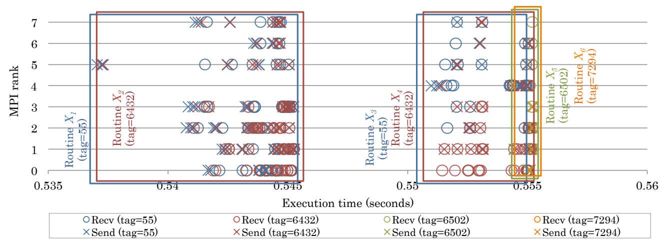

Figure 1. Communication segments in MCB (# of send messages: 205)

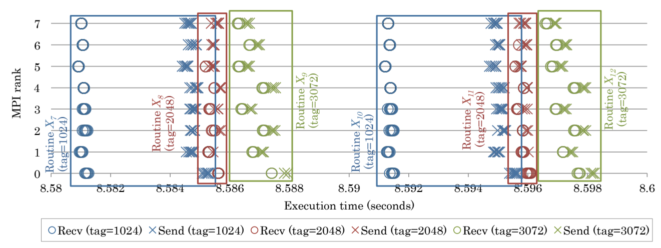

Figure 2. Communication segments in Lulesh (# of send messages: 131)

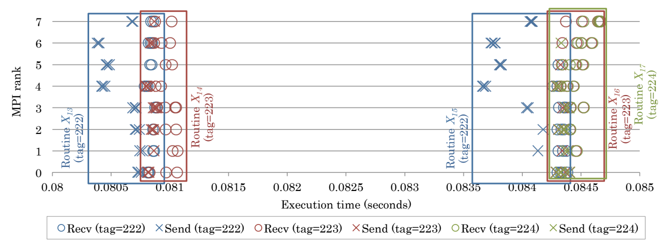

Figure 3. Communication segments in Hypre (# of send messages: 190)

3.1 Execution Model

A majority of today's large-scale parallel applications are written in the Single Program, Multiple Data (SPMD)-based message-passing paradigm. One key pattern of this paradigm is that each MPI process repetitively executes a series of *communication routines*. For example, Figures 1, 2 and 3 show this pattern in terms of receives (e.g., MPI_Recv) and sends (e.g., MPI_Send) on timeline plots of two well-known CORAL benchmark codes (MCB and Lulesh) and the Hypre library. Here, a communication routine represents a self-contained communication instance consisting of sends and receives that often use the same MPI tag and communicator for isolation.

Messages sent in a communication routine are intended to be received by the same communication routine. In our figures, we denote the communication routines as Routine $X_1, X_2 \ldots X_{17}$. If isolation is guaranteed, a communication routine can be overlapped with other communication routines for better performance. However, reasoning about this isolation can often be difficult in particular when program complexity is high, which can lead to extremely hard to debug errors.

Figure 3 shows one such bug in Hypre: the programmer did not intend the messages sent in Routine X_{15} to be received by Routine X_{13}, but there was no isolation between them. In fact, the rarely occurring hang described in Section 2.1 was caused by this unintended message matching where a message sent in Routine X_{15} was received by Routine X_{13}. Because another routine, in this case, Routine X_{14}, and other relatively intense computations occur between Routine X_{13} and X_{15}, this message race did not normally occur. However, when the code was run under significant network congestion, an overlapping of Routine X_{13} and X_{15} occurred. In order to prevent this, these two routines would have needed to use proper isolation mechanisms.

3.2 Unintended Message Race Model

Figure 4-(a) generalizes our target message race problem. Routine X consists of either multiple or none of communication routines and other computations. In the Hypre case, for example, Routine A, X and B correspond to Routine "X_{13}", "X_{14} and other computations" and "X_{15}", respectively. Even if individual routines are free of bugs, the developer must also ensure that unintended message races do not occur across different routines or different invocations of the same routine.

One mechanism to provide the necessary isolation between Routine A and B is to use unique message tags with respect to a communicator within each routine: *matching ID*. Here, each process can receive only the messages designated by the matching ID in the routine (Condition C_{msgid}). Another mechanism is to build a unique epoch (i.e., a period of logical time that *happens before* the subsequent routine) directly into a routine by means of global synchronization (e.g., `MPI_Barrier`) without in-flight messages as shown in Figure 4-(b) (Condition C_{sync}). Typically, the same communication routines are called multiple times either in an iteration or across different iterations, therefore we must ensure that condition C_{sync} is held whenever the same communication routines are called.

In general, when developing MPI applications, the programmer must ensure that either condition is satisfied, $C_{msgid} \cup C_{sync}$, to avoid message races for all communication routines. If there is any communication routine violating both C_{msgid} and C_{sync} condition, i.e., $\overline{C_{msgid}} \cap \overline{C_{sync}}$, then that the communication routines is considered *unsafe*, and the application can *potentially* encounter unintended message races if a specific timing is created. For example, Routine X_7 and X_{10} violate C_{msgid}, while C_{sync} is satisfied in Lulesh. Therefore, the routines are still *safe*. If C_{sync} is not satisfied, Lulesh can encounter the same race bug as that of Hypre's. As scientific MPI applications become more com-

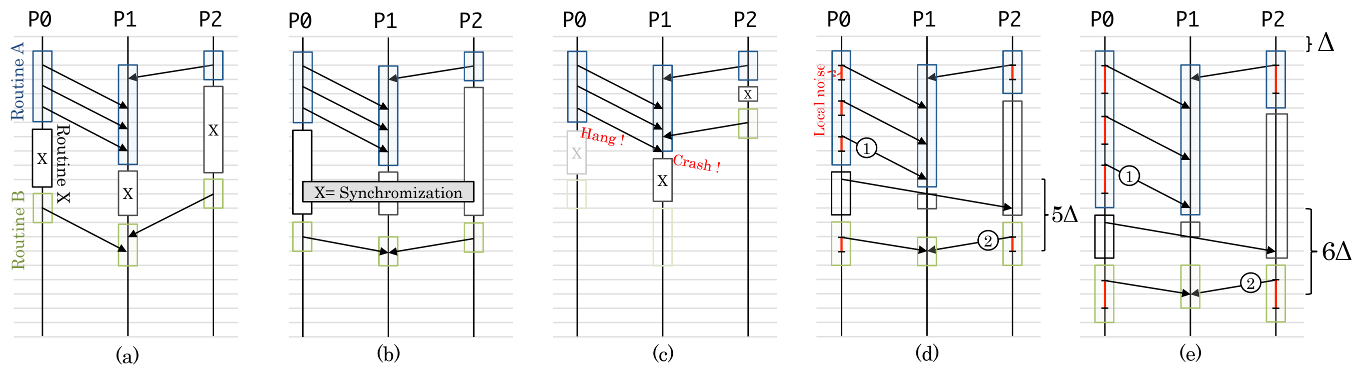

Figure 4. Message race conditions in MPI applications

plex and larger, ensuring the condition, $C_{msgid} \cup C_{sync}$, has been becoming increasingly challenging.

In this example, if Routine A and B are unsafe, then unintended message races can happen, as shown in Figure 4-(c), which can result in an application crash or hang. In practice, the manifestation of unintended message races depends on the execution time of interleaving routines (Routine X) between the two unsafe communication routines (Routine A and B). For example, if Routine X is a computation routine in a strong-scaling problem in which each MPI process operates on an increasingly smaller piece of data, a larger process count leads to a proportionally shorter execution of Routine X. Under such circumstances, message races would only begin to show up at or above a certain process count. However, the same races would not occur at small scales because Routines A and B will never be overlapped in practice as they are separated by a larger running time of Routine X.

We refer to the difficulty in reproducing unintended races due to a relatively large distance between unsafe routines as well as other relevant issues (e.g., noise effects), collectively as the *separation problem*.

3.3 Why Races Disappear under a Tool's Control?

In addition to a problem in existing debugging tools described in Section 2.2, another disadvantage of these tools is that they tend to introduce an extra runtime overhead (*node-local noise*). The occurrence of message races is highly subject to any *noise*, and therefore they can disappear due to the noise introduced by these approaches. Figure 4-(d) illustrates this problem. If processes in Routine X send and receive messages, then the node-local noise propagates through this communication, and Routine A and B can be even further separated if P2 can only transition into Routine B after receiving a message from P0 in Routine X. Nevertheless, message 2 still can arrive at P1 earlier than message 1 if message 1 is significantly delayed (Figure 5-(a)).

In practice, introducing more node-local noise separates the unsafe routines further, therefore making it harder to reproduce message races because Routine X typically contains

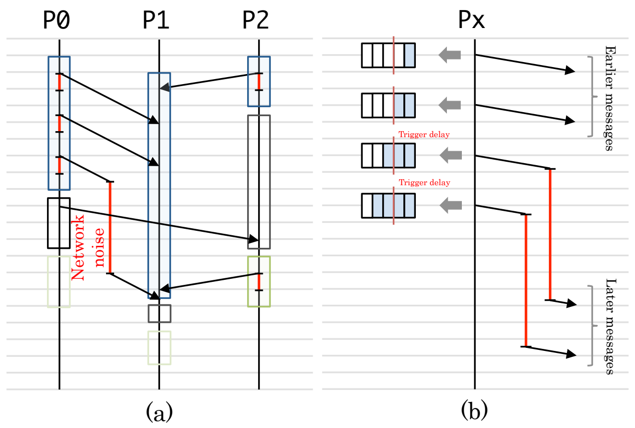

Figure 5. (a) Noise injection can overlap; (b) Noise Triggering Criteria

interleaving communications between the two unsafe routines, as shown in Figure 2 and 3. For example, if each node-local noise unit is Δ, then the interval between the arrival of message 1 and 2 is 5Δ, as shown in Figure 4-(d). However, if the node-local noise unit is 2Δ, then the interval becomes 6Δ ,as shown in Figure 4-(e).

We refer to the difficulty in reproducing this class of races due to the runtime overhead introduced by tools collectively as the *tool overhead problem*.

4. Noise Injection Techniques

When debugging message races, it is necessary to observe the bugs in debug runs. To assist programmers with observing the message races, we need practical approaches to the aforementioned problems of *separation* and *tool overhead*. Our approach is to use noise injection techniques to enable frequent manifestation of message races, and we codified this approach in a prototype called NINJA.

NINJA uses *network noise injection* techniques to control the communication behaviors of the MPI application with the goal of manifesting subtle message races more frequently. Unlike *node-local noise* (Figure 4-(d)), a right amount of *network noise* can expose unintended message races (Figure 5-(a)). NINJA provides two noise injection modes: *system-centric* and *application-centric*. In system-centric mode, NINJA induces message races by reconstructing noise signatures. In application-centric mode, NINJA first analyses the application's behavior to then directly enforce the overlapping of certain communication routines. It achieves this goal by building the application-specific knowledge into its noise injection scheme.

4.1 System-Centric Noise-Injection Mode

When injecting network noise in the system-centric mode, NINJA considers two important factors to maximize the chance for an incorrect message matching to manifest itself due to races, while limiting the application slowdown: which MPI sends to inject noise into; and how much noise to inject.

4.1.1 Noise Triggering Criteria

To induce message races in the example shown in Figure 4-(d) or (e), the ideal approach would be to delay message 1, but not message 2, as illustrated in Figure 5-(a). In general, if we delay later messages in an unsafe communication routine while not delaying earlier messages in the next unsafe routine, we can overlap two unsafe communication routines, thereby, inducing message races Figure 5-(b).

To implement such behavior, each MPI process manages a *virtual* buffer queue (VBQ) at the user level. NINJA regards MPI messages as a sequence of fixed-size chunks, i.e., packets, which are all funneled thought the VBQ of the MPI process. If the number of packets in the VBQ exceeds the configurable *VBQ threshold* (N_s), then NINJA injects a delay to all of the subsequent sends until this congestion condition is cleared. Since the buffer queue is *virtually* managed by NINJA itself, NINJA does not actually buffer the packets, but instead only keeps track of the number of packets that are expected to be in the VBQ and triggers delays as necessary.

4.1.2 Noise Amount

The second factor is the amount of each delay that should be injected. When the number of packets exceeds the threshold, NINJA computes how long the subsequent messages should wait until the VBQ is sufficiently freed up below the threshold, N_s.

More precisely, when sending an MPI message, which will be packetized into N_m packets, with the length and threshold of VBQ being configured to be N_l and N_s respectively, NINJA blocks the issuing of the send operations until N_m packets fit into the VBQ under the VBQ threshold, i.e., until $N_m + (N_l - N_s)$ packets are transmitted, as shown in Figure 6. Finally, we estimate the delayed time it takes to transmit $N = N_m + (N_l - N_s)$ packets, D, which is given

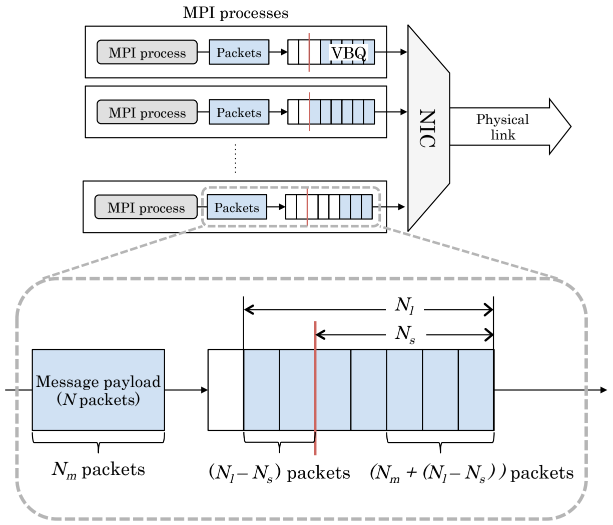

Figure 6. To enqueue all N_m packets of the message in the VBQ, N_m and $N_l - N_s$ packets must be dequeued and transmitted.

by:

$$D = \left\{ \sum_{i=1}^{N} (P_s[i]/B + C) \right\} \times S_p \quad (1)$$

where $P_s[i]$ and B are the size of the i-th packet (the 1st packet being at the head of the VBQ) and network bandwidth respectively, and S_p denotes the noise scaling factor.

This algorithm is also based on our observation that the more the system is congested and the more noisy the network, the more unintended message matchings occur (detailed results are in Figure 11 and 12). Actually, this queue model described in Section 4.1.2 and 4.1.1 is similar to general network flow controls [4, 11]: when a destination buffer do not have enough space, the congestion control engine suspends packet transmission until enough buffer space is freed up to avoid packet losses. Therefore, one can emulate more noisy environment by delaying more messages with lower VBQ threshold, N_s, and by delaying messages longer by higher noise scale factor, S_p. By using the NINJA's system-centric model, users can observe an unintended message matching more frequently than ones without using NINJA.

4.2 Application-Centric Noise-Injection Mode

The system-centric mode is useful to manifest an unintended message matching problem while minimizing overhead to the application. However, this mode cannot guarantee that all pairs of unsafe communication routines will be overlapped. As described in Section 3.2, for example, the smaller the scale is, the longer the time between two unsafe communication routines, thereby making the *separation problem* worse. For a small scale at which message races have never been observed during production runs, our system-centric noise mode will still be unlikely to manifest the race problem.

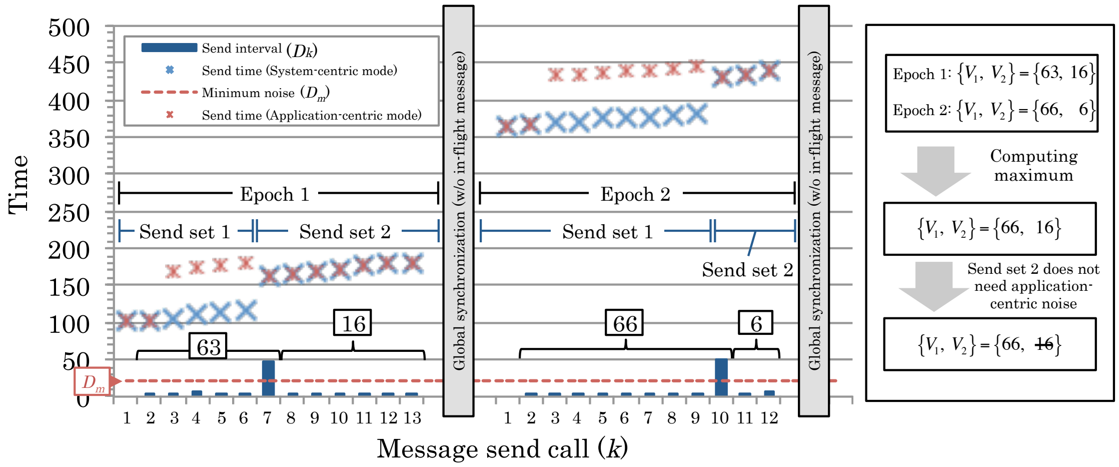

Figure 7. Example: Message send timestamps in system-centric (blue plots) and application-centric mode (red plots)

One can set a significantly large scale factor for S_p to enforce any pairs of unsafe communication routine to be overlapped. However, this configuration would introduce prohibitively large overhead to the application. In addition, debugging an application at a smaller scale is much more preferred than doing it at large scale. Thus, NINJA also provides an alternative way to enforce an overlapping between all pairs of unsafe communication routines.

4.2.1 Enforcing Message Races

To enforce an overlapping, NINJA provides an application-centric mode that uses knowledge about the application's own communication patterns even under a debugging tool's control. During a run in system-centric mode, NINJA traces and analyzes the time intervals between successive message sends, detects unsafe communications violating both C_{msgid} and C_{sync}, and finally dumps the analysis data at the end of execution (if this run in system-centric mode does not hang nor crash due to message races). During successive runs in application-centric mode, NINJA then loads the analysis data and uses this information to inject appropriate amounts of noise to enforce the overlapping of each detected unsafe communication routine.

More specifically, NINJA keeps track of injected *minimum noise* (D_m) during the system-centric mode in its the analysis file. If the time interval between two consecutive sends is larger than D_m, NINJA regards the rest of the message sends (before the next large interval is detected) as the next distinct communication routine. In Figure 7, for example, NINJA regards Send sets 1 and 2 each as a distinct communication routine and also deems that message races have not occurred between Send set 1 and 2. Because the time between Send set 1 and 2 is larger than D_m, it's not guaranteed that Send set 1 and 2 were overlapped during the run in system-centric mode.

After detecting the distinct Send sets, NINJA computes how long messages in Send set 1 should be delayed to be able to be overlapped with Send set 2 (application-centric noise amount). More precisely, NINJA computes the amount of application-centric noise for Send set i, V_i, as follows:

$$V_i = \left\{ \sum_{k=m_i}^{m_{i+1}-1} D_k \right\} \times S_a \qquad (2)$$

where D_k is the time interval between send $k+1$ and k, and m_i denotes the first message of Send set i, and finally S_a denotes an application-centric noise scaling factor. NINJA computes V_i at the end of each *epoch*. An epoch consists of a set of routines detected by our *epoch detection* algorithm. Messages sent in an epoch are solely received within the same epoch. As such, they cannot cross the epoch boundary and be received within the next epoch. Figure 7 shows an example where V_1 and V_2 are computed as 63 and 16 respectively in Epoch 1, and 66 and 6 in Epoch 2.

For the epoch detection, each MPI process counts the number of sends and receives. Whenever a global synchronization is called by the application, all of the processes call `MPI_Allreduce` to compute the global sum of the number of sends and receives. If the send count is equal to the receive count, NINJA infers that the application progressed into the next epoch because the global synchronization holds the condition C_{sync}, i.e., synchronization without in-flight messages. Whenever an epoch ends, NINJA updates V_i if new V_i is larger than old V_i to compute the maximum. By updating the maximum of V_i, NINJA ensures that it enforces an overlapping between Send set i and $i+1$ with arbitrary i in every epoch. In Figure 7, V_1 and V_2 become 66 and 16 at the end of Epoch 1.

We note that NINJA does not inject application-centric noise to the last Send set in each epoch, to avoid unnecessary noise injection: Message races will never occur between two

```
10:  int MPI_Isend (…) {
11:    if (is_delayed) {
12:      PMPI_Send_init(…);
13:      /* Enqueue request for Send thread
14:         with timer t_s */
15:    } else {
16:      PMPI_Isend(…);
17:    }
18:  }
 :
30:  int MPI_Wait (…) {
31:    while (!flag) MPI_Test(…);
32:  }
 :
50:  int MPI_Collective (…) {
51:    PMPI_Icollective(…, &reqest);
52:    MPI_Wait(&request, …);
53:  }
```

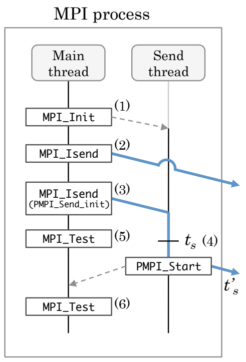

Figure 8. NINJA implementation

Send sets separated by a global synchronization. Figure 7 shows the example where NINJA injects application-centric noise by 66 time units to only Send set 1 when the VBQ gets congested at the third message. Because an application may have multiple unsafe communication routines, NINJA separately computes V_i for each message ID, i.e., a pair of tag and communicator. In addition, since each process independently computes application-centric noise, computing the application-centric noise amount minimally affects the application's scalability.

The analysis file also includes a list of message IDs violating both C_{msgid} and C_{sync}, and can be used for validating correctness of MPI programs.

5. Implementation

NINJA is implemented on top of the PMPI profiling interface, and thus can inject noise into the target application without requiring any modification to its source. In the following, we describe the implementation details of NINJA.

5.1 Send-dedicated thread

To inject network noise, we use a send-dedicated thread, one per MPI process (shown in Figure 8). In `MPI_Init`, each MPI process spawns this send-dedicated thread (*send thread*) as shown in (1) in Figure 8. The send thread is the actual initiator of the send calls for all delayed messages. When the application thread calls `MPI_Isend`, the main thread checks whether this message should be delayed based on the VBQ threshold, N_s. If the VBQ does not reach the threshold, the application thread simply calls `PMPI_Isend` ((2) in Figure 8). Otherwise, the application thread calls `PMPI_Send_init`, computes the amount of delay for either system- or application-centric noise, and then requests the send thread to call `PMPI_Start` at a scheduled send time, t_s ((3) in Figure 8). Meanwhile, the send thread periodically checks the presence of a delayed send request. If the send thread finds any delayed send request whose scheduled send time (t_s) is smaller than the current time, the send thread calls `PMPI_Start` for this message. After that, when the application thread calls one of the matching functions (e.g., `MPI_Test`) before the send thread calls `MPI_Start`, NINJA returns `flag` $= 0$ ((5) in Figure 8). If the matching function call is after `MPI_Start`, NINJA simply calls `PMPI_Test` ((6) in Figure 8). However, in most of MPI implementations, if the message payload size is less than the *eager limit*, `MPI_Test` immediately returns `flag` $= 1$ before the message is delivered to the destination. To implement this behavior, if the message payload size is less than the eager limit, the application thread copies the message payload data to its buffer space on `MPI_Isend` and returns `flag` $= 1$ even before the send thread calls `PMPI_Start`.

MPI guarantees that if an MPI process sends two messages in succession to the same destination and the two messages match the same MPI receive at the destination MPI process, then two messages are ordered, *non-overtaking*. Our network noise injector also abides by the *non-overtaking* rule. MPI also ensures that point-to-point communications within a collective call do not race with other communications. Therefore, we do not inject network noise to MPI collective calls.

5.2 Modification to Blocking Operations

In NINJA, two threads (application and send threads) concurrently call MPI functions. Thus, NINJA requires the underlying MPI implementation to support `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`. In an `MPI_THREAD_SERIALIZED` mode, NINJA locks and unlocks a mutex before and after each PMPI function call.

However, a simplistic approach can cause a deadlock. For example, consider a condition:

- An application thread of MPI process A locks a mutex in order to call `PMPI_Wait` to wait for Message B from MPI process B;
- A send thread of MPI process A has a delayed send request (Message A);
- Message B is sent after MPI process B receives Message A.

Under this condition, a deadlock occurs because a mutex is acquired by the application thread of MPI process A and the send thread is unable to grab the mutex to send Message A.

To avoid a deadlock, we re-implemented the `MPI_Wait` call family (i.e., `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome` and `MPI_Waitall`) by using the corresponding `PMPI_Test` calls (i.e., `PMPI_Test`, `PMPI_Testall`, `PMPI_Testany` and `PMPI_Testsome`). For example, the application thread periodically locks and unlocks the mutex, and the send thread can eventually acquire the lock and send Message A (Line 30-32 in Figure 8). Similarly, we also re-implement blocking collective calls by using the corresponding non-blocking collective operations (Line 50-53 in Figure 8).

Table 1. Node specification of Cab and Catalyst

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

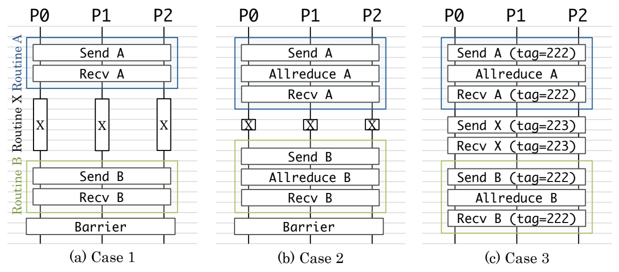

Figure 9. Case 1 and 2: Synthetic cases to capture common unintended message races. Case 3: Hypre communication pattern

6. Evaluation

In this section, we present our evaluation results showing the effectiveness and performance overhead of our injection techniques. Our evaluation is conducted on two large systems sited at Lawrence Livermore National Laboratory (LLNL): Cab and Catalyst. The main differences between two systems are: 1) Cab is larger in terms of the number of compute nodes in the machine and 2) as a production system it is more heavily utilized with more users than Catalyst. In terms of the capabilities within each node, however, Catalyst contains a higher number of compute cores than Cab (24 versus 16), and one additional InfiniBand rail that is only used for system management purposes. Taken together, applications are more subject to network congestion when they run on Cab than on Catalyst. For all our evaluation, we used MVAPICH-2.1 as the underlying MPI implementation.

6.1 System-Centric Mode

The main purpose of the system-centric mode is to inject network noise to emulate a highly congested environment for the application. Even on a quiet network system, therefore, this mode can increase the manifestation rate of unintended message races. In contrast to the application-centric mode, the system-centric mode can achieve this without obtaining the application's messaging patterns.

6.1.1 Synthetic Benchmarks For Unintended Races

To evaluate the effectiveness we created two synthetic cases capturing common race conditions (Case 1 and 2), as shown in Figure 9.

In Case 1, each process first sends messages to a randomly selected destination process (`Send A`), receives the messages from the random senders (`Recv A`), and then performs a computation followed by the same random sends and receives, i.e., `Send B` and `Recv B`. Then, each process cycles through this communications and computations for the configurable number of iterations. In Case 2, each process performs the same random sends and receives as Case 1, but this time all processes are synchronized with `MPI_Allreduce` after the random sends. In fact, this captures a common communication pattern that can emerge when each process does not know exactly from which other process(es) the messages will be sent to it. In this case, after sending messages (`Send A`), each process fills in the send array by setting the index corresponding to each destination rank, and calls `MPI_Allreduce` with `MPI_SUM` to compute how many messages should be received. Then, each process calls wild-card receives with `MPI_ANY_SOURCE` (Figure 10). This case is created to evaluate the effectiveness of NINJA when the target application is being under a debugging tool's control and hence is subject to the *tool overhead problem* described in Section 3.3. Because processes are synchronized after `Send A`, the tool's overhead introduced at `Send A` can propagate across all of the processes. Unlike Cass 1, Case 2 can decrease the frequency of exposing unintended message races.

In the experiments for manifestation of message races, we evaluate the manifestation of message races at both small and large scales. However, we only show results at a small scale, 64 processes distributed across 4 compute nodes, because of the following reasons. First, the results are same between small and large scales. Second, if we use a higher number of processes, Routine A can become more load-imbalanced, and this can cause unintended message races between Routine A and B to occur more frequently at large scale. Message races at large scale is more easy to appear than ones at small scale. Third, when debugging unintended message races, it is highly desired to be able to observe the races at small scale, as debugging is much easier and more cost-effective. In summary, we desire to evaluate the effectiveness of NINJA at small scales in more difficult and challenging scenarios to NINJA. We use 3.14 GB/s, 0.25 μsec and 1 for B, C and S_p to emulate a congested environment in system-centric mode.

6.1.2 System-Centric (S-Centric) Mode

Figures 11 and 12 show the number of iterations executed until an incorrect match occurs due to unintended message races in Case 1 and 2, respectively (a message sent in Rou-

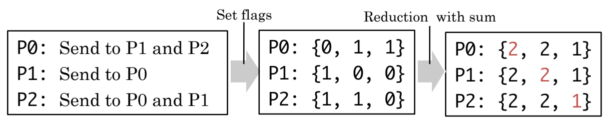

Figure 10. Typical communication patterns needed to determine the exact number of messages to receive at runtime

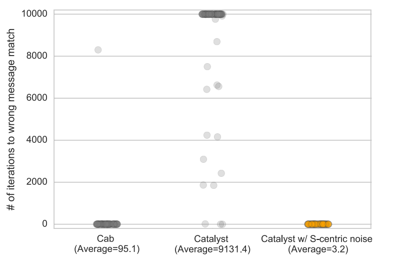

Figure 11. Case 1: # of iterations executed until an incorrect matching appears (Routine X = 1 msec computation; Max iteration = 10,000)

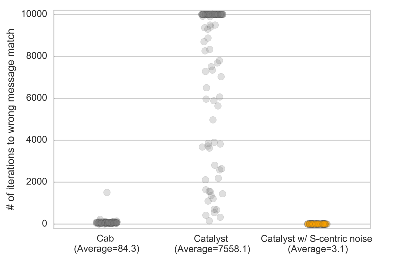

Figure 12. Case 2: # of iterations executed until an incorrect matching appears (Routine X does nothing; Max iteration = 10,000)

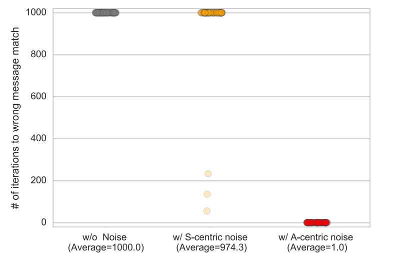

Figure 13. Case 2: # of iterations executed until an incorrect matching appears with 10 usec tool overhead in Cab (Routine X does nothing; Max iteration = 1,000)

tine B is received in Routine A). Here, we configure the maximum number of iterations to be 10,000.

Then, we run this benchmark 100 times and plot the iteration-count distributions as show in these figures. We add 1 msec computation in Routine X for Case 1 in order to emulate the phenomena described in Section 2.1, i.e., to create an scenario where unintended message races frequently manifest themselves in a congested network environment (Cab), but not in a quiet environment (Catalyst). As shown in the both figures, the incorrect message matching occurs more frequently in Cab while not appearing as easily in Catalyst. With NINJA in system-centric mode, however, we can frequently manifest the incorrect matching problem even on Catalyst with an iteration-count distribution similar to that of the experiments on Cab.

With NINJA, we also observe that the incorrect matching occurs in a few iterations. This is important because an HPC application can spend a large amount of time per iteration. The quicker the unintended message race becomes manifested, the more a programmer's productivity will increase. Our evaluation suggests that NINJA in system-centric mode not only can allow programmers to observe the bugs more frequently, but also more quickly, compared to the runs without it.

6.2 Application-Centric (A-Centric) Mode

In this evaluation, we first run our synthetic benchmarks in system-centric mode. For the cases where we cannot observe unintended message races, we use the application-centric mode by using an analysis file generated from the system-centric mode. We set the application-centric scaling factor, S_a to 1.2.

6.2.1 Overcoming Tool Overhead Problem

First, we evaluate how effectively application-centric noise can expose incorrect message matching when an amount of node-local noise akin to a debugging tool's overhead is introduced. We choose the overhead amount assuming that the programmer used `printf` statements to output necessary debug information after send, receive and matching functions complete. Thus, we add fixed 10 μsec overhead after these MPI function calls to Case 2.

Figure 13 shows the results on Cab. While message race bugs manifest themselves in the absence of tool overhead as shown in Figure 12, the same message race bugs are never captured due to the 10 μsec of tool overhead. Even in system-centric mode, the message race bugs rarely manifest themselves. However, in application-centric mode, we observe that NINJA can still frequently uncover the unintended message races. With application-centric noise, we also observe that the message race bugs typically occur at the first iteration in most of the runs. We also confirmed that the same results are achieved with even larger amounts of tool overheads: 100 μsec and 1 msec.

6.2.2 Noise Injection for Separation Problem

We also evaluate NINJA on the separation problem described in Section 3.2. For this evaluation, we add 1 msec computation in Routine X. Figure 14 shows the results. Similarly to Figure 13, the application-centric mode can efficiently uncover the message race bugs even if we introduce a far worse separation problem than the original condition. Because the application-centric mode enforces an overlapping between two unsafe communication routines by tailoring the delays to the application, this mode can even more frequently uncover the unintended message races within the application.

6.2.3 Real-World Bug in Hypre 2.10.1

Finally, we apply NINJA to a known bug in Hypre-2.10.1. Hypre is a library for solving sparse linear systems of equations for MPI applications. A non-deterministic message race bug exists in one solver that computes a sparse approximate inverse pre-conditioner. We use an example code of the solver, `ex5.c`, in the Hypre-2.10.1 package. Case 3 in Figure 9 depicts the two unsafe communication routines that are used in this solver based on Figure 3. Although this solver calls a global synchronization (`Allreduce A`) before receive calls (`Recv A`), an unintended message matching can

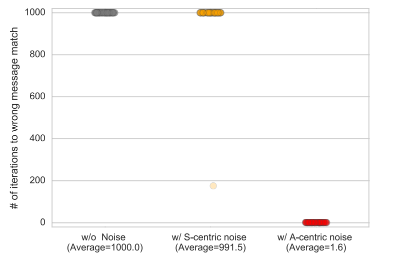

Figure 14. Case 2: # of iterations executed until an incorrect message matching occurs in Cab (Routine X = 1 msec computation; Max iteration = 1,000)

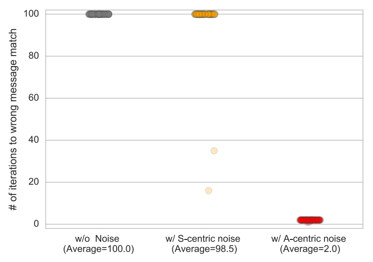

Figure 15. Case 3: # of iterations executed until an incorrect message matching appears (Max iteration = 100)

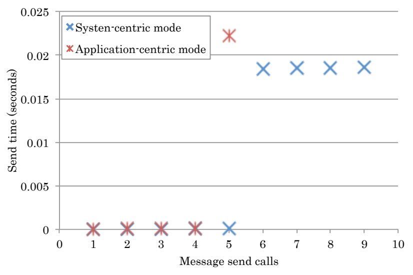

Figure 16. Case 3: Send time in system-centric vs. application-centric mode

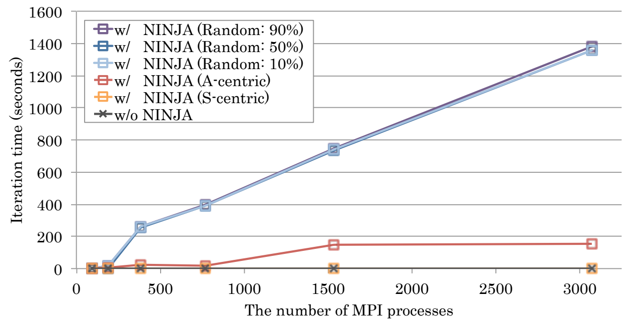

Figure 17. A single iteration time

still occur between Routine A and B because C_{sync} is not held. However, this race infrequently occurs in practice because there exist several communication and computations routines, significantly separating these two unsafe communication routines. Because of this pattern, the message race bugs in Hypre have not been fixed during the 14 years of its history. Figure 15 shows the results. Similarly to the results in Figure 13 and 14, NINJA can successfully uncover this race at small scale in application-centric mode.

Figure 16 shows the actual send time profile of one of the MPI processes, contrasting system- and application-centric modes. The first set of sends belongs to Routine A, and the second set of sends to Routine B. We normalize the send time of these modes such that the time stamp of the first send is 0. As shown in this figure, NINJA injects sufficient amounts of delay to overlap Routine A with B.

6.3 NINJA Performance Overhead

NINJA can expose unintended message races by injecting network noise. One could attempt to achieve the same objective by injecting large amounts of noise to randomly selected messages in order to overlap all pairs of unsafe communication routines. However, the naive random approach can introduce a large runtime overhead to applications with no clear notion of *target communication patterns*. In contrast, NINJA's targeted injection schemes do not incur high overheads.

Figure 17 shows the average iteration time of running the Hypre code with or without NINJA (Random, System-centric and Application-centric noise) as shown in Table 2. We can observe that a random noise injection scheme significantly impacts the code's running time. On the other hand, the application-centric mode injects noise to only unsafe communication routines with noise amount just good enough to expose a race. This limits the performance impact to the application. Further, the system-centric mode only emulates a congested network environment with very small network noise, and thus its overhead is even smaller to the point where it is negligible. Although the system-centric mode cannot provide a guarantee to overlap all pairs of unsafe communicating routines, this mode is useful if one wants to test a large number of codes with bare-minimum overhead for message races.

7. Related Work

Debugging non-deterministic bugs is a challenging task, since these bugs can manifest themselves in production runs, but may disappear in debugging runs. With trends towards asynchronous communications with wild-card receives, MPI applications are becoming increasingly complex and non-deterministic. Debugging tools are critical to observing, finding and fixing bugs, but most of the exiting debugging tools cannot cope well with unintended message races as they cannot help much if the bugs do not occur under their control.

MPI formal verifiers such as ISP [24] and DAMPI [25] explore all possible message-receive patterns so that the tools can eventually hit a target bug. However, these tools are not practical for large-scale applications because they need to run for a significantly large number of iterations to examine all possible message interleavings until a message race is hit. In contrast, NINJA prioritizes examining communication patterns for overlapping two unsafe communication routines

Table 2. NINJA noise trigger, noise amount and target message ID

	Random noise	System-centric	Application-centric
Noise trigger	90, 50, 10% of messages	Method described in Section 4.1.1	Method described in Section 4.1.1
Noise amount	Iteration time	Formula 1 (in Section 4.1.2)	Formula 2 (in Section 4.2.1)
Target matching ID	All	All	Only matching ID in unsafe communication routines (Section 4.2.1)

so that the unintended message races can manifest within a few iterations, which is a more practical solution for large-scale applications.

Record-and-replay is an attractive approach that can ease the efforts needed to debug non-deterministic bugs, as the tool can repeatedly reproduce incorrect message matching once it observes and records this misbehavior. Mainly, two common approaches, data-replay and order-replay, exist: Data-replay [2, 3, 17] records all the received messages including the message receive order and payloads. With data-replay, programmers can replay the buggy behavior with a single MPI process. An alternative approach is order-replay [14–16, 20]. Order-replay only records message-receive order. Although order-replay requires programmers to run the application at the identical number of process count for replay as the record run, the approach can significantly reduce tool overhead. However, both approaches can only deterministically replay incorrect message matching only when the unintended message races manifest themselves during the record phase. By contrast, NINJA can significantly help exposing these errors even with the recording overhead. Hence, NINJA complements existing record-and-replay approaches.

There are several data race detection techniques in shared-memory [6, 18, 21, 22]. The lockset algorithm in Eraser [21] and Active-testing [18] analyzes mutex-lock/unlock behaviors. State-of-the-art data-race detectors such as FastTrack [9] and ThreadSanitizer [22] exploit *happens-before* relations around the lock/unlock concept. However, techniques to analyze and control schedules to uncover unintended races are fundamentally different between the message-passing and shared-memory paradigms. Events in shared-memory, such as memory reads/writes, occur in a single-memory space, whereas message sends/receives occur in distributed-memory spaces. Because of these differences, data-race detection techniques cannot be directly applied to message races. Delay-bounded scheduling explores possible task/thread schedules until buggy behaviors are hit. However, simply delaying MPI processes using such a technique would have a similar effect as injecting node-local noise. Node-local noise does not always work as described in Section 3.3. Instead of node-local noise, NINJA employs a novel network noise injection technique to the message-passing paradigm. To the best of our knowledge, no data-race analysis technique delays memory accesses at the level of channel-to-memory itself.

There exists a strong body of work on the effects of system and network noise [1, 7, 8, 13, 23]. Ferreira et al. [8] developed a noise injection functionality at the OS level, and other works [1, 7, 23] inject noise at the user level. However, these approaches are only capable of injecting OS jitters, i.e., node-local system noise. To adequately delay message sends for our purposes, network noise is necessary. Hoefler et al. [13] investigated the impact of network noise to the application's *performance* by using a *simulator*. In contrast, we developed a *noise injection tool using PMPI* for *debugging* message race bugs. To the best of our knowledge, our work (the NINJA network-noise injection tool) is the first application of network noise to uncover correctness problems in MPI applications.

8. Conclusions

An ability to observe non-deterministic bugs is the critical prerequisite to finding and fixing them. To assist in this, we developed a novel noise injection technique (NINJA) that can uncover subtle and untended message races based on comprehensive analysis on non-deterministic MPI applications. NINJA injects network noise into the application execution in order to expose message race bugs more frequently and quickly during debugging runs. Our evaluation showed that NINJA is highly effective to expose bugs in representative synthetic benchmark codes as well as a real-world bug within Hypre, one of the mostly widely used sparse matrix solver libraries.

NINJA has a promising future to support programming models beyond MPI, which include tasking models. Because non-deterministic bugs often occurs as a function of specific timings of parallel interaction, we believe that our approach can be extended and generalized for other programming models. Indeed, that is a significant part of our future direction.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by LLNL under contract DE-AC52-07NA27344 (LLNL-CONF-699343).

References

[1] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–12, Sept 2006. .

[2] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine*

and Message Passing Interface, volume 4757 of *Lecture Notes in Computer Science*, pages 297–306. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75415-2. . URL `http://dx.doi.org/10.1007/978-3-540-75416-9_41`.

[3] C. Clemencon, J. Fritscher, M. Meehan, and R. Ruhl. An Implementation of Race Detection and Deterministic Replay with MPI. In *EURO-PAR '95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 155–166. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60247-7. . URL `http://dx.doi.org/10.1007/BFb0020462`.

[4] D. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-470154-2.

[5] CORAL. Collaboration of Oak Ridge, Argonne, and Livermore benchmark codes. `https://asc.llnl.gov/CORAL-benchmarks`.

[6] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 411–422, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. . URL `http://doi.acm.org/10.1145/1926385.1926432`.

[7] C. Engelmann. Investigating operating system noise in extreme-scale high-performance computing systems using simulation. In *Proceedings of the http://www.iasted.org/conferences/home-795.html* 11^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2013*, Innsbruck, Austria, Feb. 11-13, 2013. http://www.actapress.comACTA Press, Calgary, AB, Canada. ISBN 978-0-88986-943-1. . URL `http://www.christian-engelmann.info/publications/engelmann12investigating.pdf`.

[8] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008. .

[9] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. . URL `http://doi.acm.org/10.1145/1542476.1542490`.

[10] M. P. Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994. URL `http://www.mpi-forum.org/`.

[11] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato. Congestion control in infiniband networks. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, pages 158–159, Aug 2005. .

[12] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. Runtime error detection with must: Advances in deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389037`.

[13] T. Hoefler, T. Schneider, and A. Lumsdaine. The impact of network noise at large-scale communication performance. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009. .

[14] J. C. d. Kergommeaux, M. Ronsse, and K. D. Bosschere. MPL*: Efficient Record/Play of Nondeterministic Features of Message Passing Libraries. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 141–148, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66549-8. URL `http://dl.acm.org/citation.cfm?id=648136.746462`.

[15] D. Kranzlmüller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In P. Zinterhof, M. Vajteršic, and A. Uhl, editors, *Parallel Computation*, volume 1557 of *Lecture Notes in Computer Science*, pages 490–499. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-65641-8. . URL `http://dx.doi.org/10.1007/3-540-49164-3_47`.

[16] D. Kranzlmüller, C. Schaubschläger, and J. Volkert. An Integrated Record & Replay Mechanism for Nondeterministic Message Passing Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 192–200. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42609-7. . URL `http://dx.doi.org/10.1007/3-540-45417-9_28`.

[17] R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debugging Message-passing Parallel Programs. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 502–511, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-2630-5. URL `http://dl.acm.org/citation.cfm?id=147877.148058`.

[18] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 51:1–51:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. . URL `http://doi.acm.org/10.1145/2063384.2063452`.

[19] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park. *MPIRace-Check: Detection of Message Races in MPI Programs*, pages 322–333. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-72360-8. . URL `http://dx.doi.org/10.1007/978-3-540-72360-8_28`.

[20] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz. Clock delta compression for scalable order-replay of non-deterministic parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 62:1–62:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. . URL `http://doi.acm.org/10.1145/2807591.2807642`.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-

threaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997. ISSN 0734-2071. . URL `http://doi.acm.org/10.1145/265924.265927`.

[22] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. . URL `http://doi.acm.org/10.1145/1791194.1791203`.

[23] G. Shipman, P. M., Cormick, K. Pedretti, S. Olivier, K. B. Ferreira, R. Sankaran, S. Treichler, A. Aiken, and M. Bauer. Analysis of application sensitivity to system performance variability in a dynamic task based runtime. In *The Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures*, 2015.

[24] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical mpi programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 261–270, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. . URL `http://doi.acm.org/10.1145/1504176.1504214`.

[25] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. . URL `http://dx.doi.org/10.1109/SC.2010.7`.

[26] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. Mpiwiz: Subgroup reproducible replay of mpi applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 251–260, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. . URL `http://doi.acm.org/10.1145/1504176.1504213`.