Improving Distributed Memory Applications Testing By Message Perturbation

Richard Vuduc, Martin Schulz,
Dan Quinlan, Bronis de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
7000 East Avenue, Livermore, CA USA
{richie,schulzm,dquinlan,bronis}@llnl.gov

Andreas Sæbjørnsen
Department of Physics
University of Oslo
Oslo, Norway
andsebjo@student.matnat.uio.no

ABSTRACT

We present initial work on perturbation techniques that cause the manifestation of timing-related bugs in distributed memory Message Passing Interface (MPI)-based applications. These techniques improve the coverage of possible message orderings in MPI applications that rely on nondeterministic point-to-point communication and work with small processor counts to alleviate the need to test at larger scales. Using carefully designed model problems, we show that these techniques aid testing for problems that are often not easily reproduced when running on small fractions of the machine.

Our perturbation layer, JitterBug, builds on P^{N}MPI, an extension of the MPI profiling interface that supports multiple layers of profiling libraries. We discuss how JitterBug complements existing MPI checking tools through the P^{N}MPI framework. We present opportunities to build additional tools that statically analyze and directly transform the source code to support testing and debugging MPI applications at reduced scale.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*parallel programming, distributed programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*distributed debugging, testing tools, tracing*

General Terms

Algorithms, Measurement, Reliability, Experimentation

Keywords

MPI profiling, interposition layer

1. INTRODUCTION

Large-scale scientific applications using the Message Passing Interface (MPI) [11] are often difficult to test. Systems at U.S. Department of Energy laboratories use tens of thousands of processors, and systems will soon have hundreds of thousands to millions of processors. The development environment for these systems is typically a small fraction of the whole machine and tools are often difficult to use at larger sizes. However, applications may exhibit errors, such as deadlocks and races, that are sensitive to timing and message orderings that only occur in the real execution environment. We seek methods that can cause these bugs in the development environment.

In this paper, we evaluate techniques that improve testing coverage for bugs in MPI applications that rely on nondeterminism, one of the most important classes of bugs to developers [3]. Nondeterministic operations include MPI "receive-any" calls, in which the receiving process accepts a message from any sender instead of a specifically identified source. A developer often uses these calls for performance reasons (*e.g.*, to tolerate high variances in message latencies), but may have a bug that depends on the order of received messages. Users typically must use repeated testing to uncover these bugs. However, they can only hope that buggy orderings are covered by even a huge test count.

The difficulty of finding nondeterministic MPI bugs is exacerbated by machine-dependent idiosyncracies [9]. Consider a simple MPI application run with p processes in which $p-1$ processes each send 1 message to a master process, and the master process executes $p-1$ receive-any operations for those messages. For a $p = 8$ process run, Figure 1 (left) shows which of the $(p-1)! = 5040$ possible message orderings, linearized in increasing lexicographic order by process rank along the y-axis, occurs in 5040 consecutive trials (x-axis) of this program. After all trials, we observe only about 20% of the possible orderings; if all permutations had been equally likely, we would expect to see 63% instead (see Section 2). These orderings are biased toward those in which the first message arrives from either process 1 or process 2, as indicated by the relatively dense bottom two bands. Although repeated test executions may cover only a fraction of the possible cases, we could improve the coverage by inserting random delays at send operations, or even explicitly buffering and reordering all receive operations. Our MPI send and receive perturbation modules do just that, improving the ordering coverage as shown in Figure 1 (right).

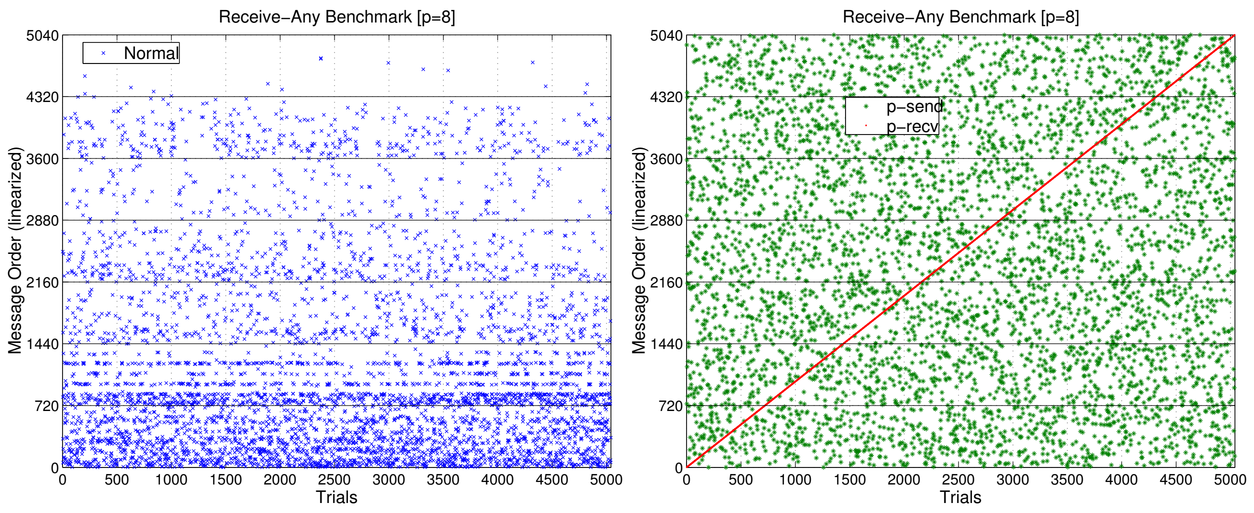

Figure 1: (Left) Machine-dependent bias in occurrence of message orderings. (Right) Improved the coverage of possible orderings by random delays of sends to produce a uniform distribution of orderings, or explicit buffering and deterministic reordering of receives (diagonal line).

The techniques we present are based on the idea of controlling or otherwise influencing the message order to improve the coverage of possible message orderings during testing (Section 3). These perturbative techniques apply and extend ideas proposed in the IBM ConTest tool for multithreaded applications [4] to the distributed memory MPI case. We consider perturbing both send and receive operations. We use carefully designed model problems that abstract the patterns appearing in large-scale scientific applications to show how to cover significantly more message orderings in order to find bugs during testing.

We are implementing these ideas in JitterBug, an MPI layer built using P^{N}MPI [14]. P^{N}MPI extends the MPI profiling interface (known as PMPI) to support multiple layers of profiling libraries, which run transparently to the user. Since perturbation-induced errors typically appear as mismatched send/receive operations and other types of deadlocks, JitterBug should be used with other MPI checking tools, such as Umpire [15], MARMOT [8], or the Intel Message Checker [3]. We are pursuing the combined uses of these tools for evaluation on full-scale applications.

2. NONDETERMINISM IN MPI: MODEL PROBLEMS

We describe several model problems that illustrate the kinds of bugs our perturbation techniques address. Though all are artificial, three are coarse abstractions of realistic application behavior which we can tune to control the difficulty of finding a bug. We empirically evaluate the effectiveness of message perturbation for these problems in Section 4.

The model problems all use the MPI feature that allows a process to receive messages from any node, without having to know the sending node *a priori*. MPI one-sided operations, such as MPI_Put and MPI_Get, provide shared-memory style operations for reading and writing directly from locations in another process's address space [12]. Since 1-sided communication has a ready mapping to the shared-memory multithreaded case, we do not consider it here.

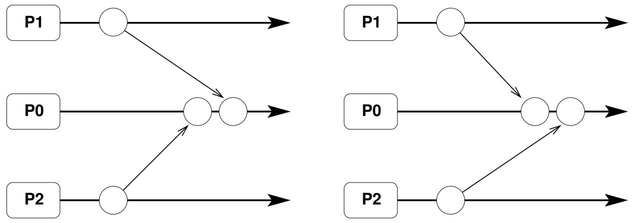

Figure 2: Example of nondeterministic receives and possible consequences.

```
1: Let $r$ = rank of this process
2: Let $p$ = the number of processes
3: if $r == 0$ then /* Master receive on process 0 */
4:   for all $n \in \{1, 2, \ldots, p-1\}$ do /* Neighbors $n$ */
5:     MPI_Recv($*$); /* Receive-any */
6: else /* Send message to process 0 */
7:   MPI_Send(message $\rightarrow$ 0);
```

Figure 3: Nondeterministic receives: Process 0 receives 1 message from every other process in any order.

2.1 Nondeterministic receives

Applications that use the receive-any feature can exhibit nondeterministic behavior. Since parallel processes are not synchronized, multiple concurrent messages matching a single receive can be seen by the application in any order. Figure 2 shows an example: the receive operation in process 0 matches both messages sent by process 1 and process 2. The order in which process 0 receives these messages and, as a result, the value that it computes depends on the timing.

We consider the p-process model problem shown in Figure 3, in which a single process performs $p-1$ receive-any operations to collect one message from each of the other $p-1$ processes. The two possible executions of Figure 3 for $p=3$ appear in Figure 2. Kranzlmüller and Schulz show that the

```
 1: Let $r$ = rank of this process
 2: Let $p$ = number of processes
 3: Let $M$ = largest value the type can hold
 4: Let $u$ = any constant $\geq 1$
 5: if $r == 0$ then /* Master process */
 6:   Let $s = M - \left(\frac{p}{2} - 1\right) * u$
 7:   for all $n \in \{1, 2, \ldots, p-1\}$ do /* Neighbors $n$ */
 8:     $t \leftarrow$ MPI_Recv($*$); /* Receive-any */
 9:     if $(M - s) < t$ then /* $s + t$ overflows */
10:       print ("Failure");
11:     $s \leftarrow s + t$;
12: else
13:   if $r$ is odd then
14:     Let $n = u$;
15:   else
16:     Let $n = -u$;
17:   MPI_Send($n \rightarrow 0$); /* Send $n$ to master */
```

Figure 4: The overflow model problem has the same communication pattern as the nondeterministic receive benchmark (Figure 3), but fails under some message orderings.

nondeterministic behavior of this problem varies drastically across architectures and none of the test architectures exhibits behavior even close to complete coverage of all possible message permutations [9]. Hence, any attempt to debug such codes will only reach a subset of possible executions, leaving the possibility for undetected race conditions.

In this model problem, the receiving process sees up to $d = (p-1)!$ possible message orderings. If all orderings are equally likely, then in k consecutive executions, the receiver should expect to see C unique orderings, where

$$C = d - d \cdot \left(1 - \frac{1}{d}\right)^k \tag{1}$$

When $k = d$ and $d \gg 1$, $C \approx d \cdot \left(1 - \frac{1}{e}\right)$, or roughly 63% of d. We revisit Equation 1 during the analysis of experimental results in Section 4.

The nondeterministic receive benchmark of Figure 3 does not have any bugs in it, so we introduce the *overflow model problem*, which has the identical communication pattern but also has a subtle bug, in Figure 4. A master process computes the sum s of contributions from all other processes, where processes with odd rank send the value u, while even-ranked processes send $-u$. The final sum should not overflow, but the partially accumulated sum could overflow the finite precision of s under some message orderings. We can control how commonly this bug might appear by an appropriate choice of u. The overflow problem models, for instance, a single iteration of a Monte Carlo-based numerical algorithm.

2.2 Race-addition problem

Many multithreaded synchronization bugs have direct message passing analogues. We present one for a simple non-atomic operation race condition example from a paper about using perturbation techniques with the ConTest tool to find multithreaded application bugs (Figure 3 in Edelstein, *et al.* [4]). Figure 5 shows MPI pseudocode for our analogue.

The program in Figure 5, when run with $p = 6$ processes, will print the value, "11111" if all messages are received

```
 1: Let $r$ = rank of this process
 2: Let $p$ = number of processes
 3: if $r == 0$ then /* Master process */
 4:   sleep ($\frac{1}{10}$ second);
 5:   Let $s = 0$
 6:   while receive buffer is not empty do /* Sum values
      from all others */
 7:     $s \leftarrow s +$ MPI_Recv($*$); /* Receive-any */
 8:   print (s);
 9: else
10:   Let $n = 10^{r-1}$;
11:   MPI_Send($n \rightarrow 0$); /* Send $n$ to master */
```

Figure 5: Pseudocode of the race-addition model problem, adapted from a ConTest multithreaded example to MPI.

within $\frac{1}{10}$ of a second ("sleep" statement); otherwise, another value will be printed and a bug will occur due to one or more unmatched sends. In the absence of heavy cluster utilization or network traffic, the time to send a message is typically very short compared to the sleep time in this example, and so this bug will occur seldomly. The goal of our perturbation techniques, as with ConTest, is to help elicit the buggy behavior in which all possible 2^{p-1} values are printed.

2.3 Dining philosophers

The multi-process synchronization errors that occur in multithreaded shared memory applications also occur in distributed memory codes. A classic abstraction of such errors is the well-known dining philosopher problem. In this problem, p philosophers sit around a circular table, separated from one another by one of p utensils. Each philosopher alternates between "thinking" and "eating" states. Before eating, each philosopher must acquire both utensils (one to his left and the other to his right). However, he can only acquire one utensil at a time and does not relinquish his utensils until he has finished eating. The problem is to devise a protocol for acquiring utensils that avoids starvation and deadlock. In distributed memory codes, this kind of problem arises in applications that use distributed data structures, shared files, or that use MPI one-sided communication.

The symmetric philosopher protocol, in which each philosopher first attempts to acquire the utensil to left, and when successful, then proceeds to acquire the utensil to his right, will eventually deadlock. However, such a bug may be unlikely to occur if, say, the thinking time and/or eating times are large compared to the time to acquire both utensils. To simulate such conditions in our experiments, we consider an implementation of the symmetric protocol in which we prescribe the distributions of time in the thinking and eating states. In this way, we can control the probability that the deadlock error occurs in the symmetric protocol by adjusting these times to be large relative to the acquisition time in our particular network.

Given an even number $p \geq 4$ of processes, we consider an implementation of the dining philosophers problem in which the even numbered processes are philosophers, and the odd numbered processes are utensils. A philosopher acquires utensils by sending a message to each of his utensil neighbors, and the utensils respond with a flag letting the

```
 1: Let $r$ = rank of this process
 2: Let $p$ = the number of processes
 3: Let $N$ = a neighborhood $N \subseteq \{1,\ldots,p\}\backslash r$
 4: repeat /* Until convergence */
 5:    $s \leftarrow$ local_computation();
 6:    for all $n \in N$ do /* Broadcast to neighbors */
 7:       MPI_Isend($s \rightarrow n$);
 8:    for all $n \in N$ do /* Receive from neighbors */
 9:       $x \leftarrow$ MPI_Recv($*$); /* Receive any */
10:       $s \leftarrow s \oplus x$; /* Combine */
11:    MPI_Waitall();
12: until cond($s$)
```

Figure 6: Convergence loop example: A send-recieve mismatch error may occur if $\oplus$ is non-associative.

philosopher know whether acquisition was successful or not. A philosopher continues to request utensils until both are acquired.

2.4 Convergence loop

We next consider a situation in which all processes first redundantly evaluate some condition computed from global state and then take the same action based on the result. A bug may occur if the condition depends on the order of arriving messages that contain the global state. Figure 6 shows a common pattern in scientific applications. Each process r among a total of p processes concurrently executes a loop that repeatedly computes some value s locally, then performs a reduction operation among some set N of neighboring processes in lines 6–11 (*e.g.*, a residual calculation involving a dot product on distributed vectors), and finally evaluates some condition cond(s) on the result. All processes should take the same action in line 12 (*e.g.*, all repeat if the residual is large, or exit the loop if it is small), as an unmatched send-receive error would occur otherwise.

In Figure 6, we implement a reduction operation consisting of non-blocking sends (lines 6–7) and receive-any calls (lines 8–9). This implementation is efficient if there is some load imbalance or otherwise high variance in the latency of communication among the neighbors since messages may be received in any order (line 9). However, if the $\oplus$ operator is non-associative—for example, $\oplus$ is floating-point addition—an error may occur. We can control the likelihood of an error occurring by choosing cond(s) appropriately.

Our implementation of this abstract problem computes a global all-to-all sum, using single-precision IEEE floating-point addition as the non-associative operator, $\oplus$.[1] At line 5, each process performs a simple assignment of a fixed, carefully chosen constant. In principle, all processes should compute the same sum at each iteration. However, each sum will differ in practice due to round-off errors, since each process could combine the local sums from the other processes in $(p-1)!$ ways depending on the order of arriving messages at the receive-any calls. Since there will be some distribution of possible values of s, we can implement cond(s) as a simple test of whether s is less than some value chosen with respect to this distribution, thereby allowing us to control the relative frequency of the processes disagreeing.

[1]Special compiler flags ensure that we do not use the extended precision registers to compute the intermediate sums.

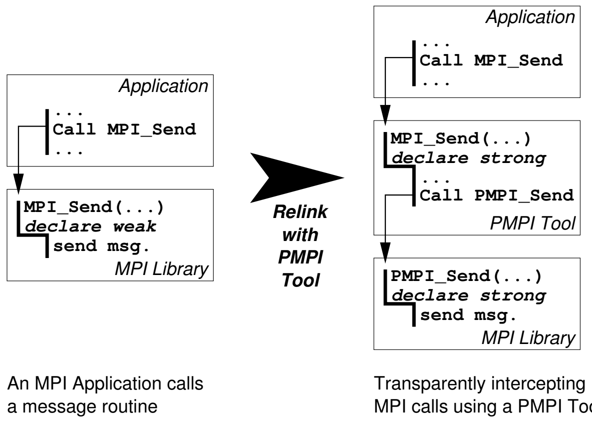

Figure 7: The MPI Profiling Interface PMPI.

3. A PERTURBATION LAYER FOR MPI

Edelstein, *et al.*, propose automated introduction of random delays, or *irritators*, at synchronization and shared-memory access points in multithreaded programs [4]. These delays increase the number of thread interleavings at these critical points of program execution, thereby improving the quality of testing. As Figure 1 and the model problems of Section 2 suggest, the same idea applies to MPI applications at points of nondeterminism.

3.1 MPI perturbation techniques

When implementing similar techniques for MPI we distinguish two main approaches: a) sender-side and b) receiver-side perturbation. The former delays each send by a random amount of time, while the latter buffers incoming message at nondeterministic receives and reorders their delivery to the application.

Both have advantages and disadvantages. Sender-based perturbation is simple to implement and models typical perturbations encountered on real machines. However, it does not guarantee that messages will actually be reordered. Receiver-based perturbation, on the other hand, may actively reorder messages and includes the ability to compute coverage. However, receiver-based techniques are complex to implement since they must store and retrieve complete messages with arbitrary datatypes.

In this initial study, we therefore concentrate on sender-based perturbation as well as a simplified receiver-based perturbation based on basic datatypes.

3.2 Implementation

The MPI Standard defines a mechanism, the PMPI interface, to intercept any MPI call using a transparent interposition layer [11]. Tool developers implement new versions of all targeted MPI routines and then use a function-shifted interface to invoke the new layer. Figure 7 illustrates a common method to implement this interface: weak symbols. When linked with the application, a tool then interposes itself between the application and the MPI library and thereby ensures the execution of the tool on every redefined MPI call. We use this interface to implement both sender- and receiver-side perturbation by intercepting all send or receive calls respectively.

On the sender-side, our simple perturbation module delays the send by an amount of time chosen uniformly at random from some user-specified interval, $[0, t_{\max}]$. In general, the user should choose $t_{\max}$ to match the characteristics of her application and machine. If $t_{\max}$ is too short, the perturbation will not result in the desired effect of creating more execution orders. If instead it is too long, the application is needlessly stalled.

We have also implemented an experimental option to enable the module to select $t_{\max}$ automatically by tracking a small window of the times between recent send calls, and setting the delay-time in proportion to the average of these times. We evaluate this module on the dining philosophers example in Section 4.3.

On the receiver-side, we implement a simple, purely deterministic protocol in the perturbation module. Our module assumes at most 1 message from all other processes at each receive-any call-site, buffers a fixed number of messages at each call-site and returns these messages in permuted order. For each batch of receive-any calls, the permutation is advanced purely deterministically in lexicographically increasing order. This behavior ensures full coverage of message orderings after $(p-1)!$ trials for any message pattern matching the nondeterministic receive model problem (Section 2.1). We are extending the receiver-side module to be more flexible with respect to other message patterns.

A perturbation layer alone, however, is insufficient to help programmers write more portable and correct MPI code. We must combine it with other tools that check the execution for certain properties. Unfortunately, PMPI allows only a single tool to add interposing code into the MPI layer: the link and naming structure of PMPI does not support multiple tools concurrently.

We overcome this obstacle through P^{N}MPI, a tool infrastructure developed at LLNL to load and combine multiple PMPI modules dynamically [14]. We outline the architecture of P^{N}MPI in Figure 8. It intercepts all MPI function calls, loads any user requested PMPI tool modules dynamically into the application, and then creates a call chain for each MPI call through all loaded PMPI modules. Thus, we can invoke several independent PMPI tools during a single run of the MPI application.

4. EXPERIMENTAL RESULTS

We implemented the model problems of Section 2 and the perturbation modules of Section 3. In this section, we evaluate the perturbation techniques experimentally, organizing the presentation by problem. Collectively, the results support the utility of message perturbation.

We performed these experiments on a tightly-coupled Linux cluster consisting of 1152 two-way 2.4 GHz Pentium 4 nodes, Quadrics Elan3 interconnect, and Quadrics MPI. The MPI point-to-point latency is approximately 5 μs, and the peak point-to-point bandwidth is approximately 340 MB/s. In these proof-of-concept experiments, we use at most 8 nodes and always use only 1 CPU per node. We consider three run modes in these experiments: without any perturbation ("normal"); with simple sender-side perturbation ("p-send"); and with receiver-side perturbation ("p-recv").

4.1 Nondeterministic receives

We explore the coverage of all possible message orderings of the nondeterministic receive problem (Section 2.1) under all three run modes. We experimented with several maximum sender perturbation delays for the p-send mode. We present results for a small maximum delay time (roughly 500 μs) since it yielded a good approximation to the theoretical case of a uniform distribution of message orderings, where all orderings are equally likely.

We show the empirical cumulative distribution function of possible message orderings in Figure 9 (left). A uniform distribution is shown for reference as the diagonal line. The normal mode exhibits a strong bias toward message orderings in which the first arriving message comes from either process 1 or process 2 (2 left-most vertical bands, which correspond to the lower horizontal bands in Figure 1). These orderings account for 70% of all normal observations. By contrast, p-send approximates a uniform distribution.

We plot the number of unique orderings observed as a function of the number of trial executions in Figure 9 (right). In this figure, each experiment consisted of $(p-1)! = 5040$ trial runs of the model problem. We repeat the experiment 100 times for each run mode. We plot the mean (markers) and the maximum and minimum (solid lines around means) percentage of possible unique orderings observed after a given fraction of the trial runs. After $(p-1)!$ trials, we observe only 20% of the total possible orderings in the normal case, compared to 63% for sender-side perturbation. Indeed, the curve for sender-side perturbation nearly exactly matches what we would expect from Equation 1, plotted as the "uniform distribution" line in Figure 9 (right).

Figure 9 shows that the p-recv mode achieves the highest coverage, but this observation obscures an important issue for testing. Our p-recv implementation covers all orderings, but chooses them in a biased, purely deterministic fashion. Contrast this behavior to a hypothetical "ideal" perturbation process in which, at each trial, we select a message ordering uniformly at random without replacement. Our p-recv will not approximate this ideal at trial counts significantly less than required for exhaustive coverage.

We have observed anecdotally in separate runs (not shown) that the "normal" case is sensitive to the prevailing load conditions on the cluster. Although the nodes were dedicated to our runs, the network is shared among a large number of users. This network traffic lends a helpful perturbative effect. In fact, Figure 9 shows the results under moderately heavy usage; under lightly loaded conditions, we observed the coverage in the normal case to be as low as 5–10% for $p = 8$ after 5040 trials. More comprehensive runs, taken many times a day over some period of time, would be needed to build better statistics.

Greater message ordering coverage improves the ability to detect the bug in the overflow problem of Figure 4. This overflow problem will normally not fail, but when the contributions are returned in such a way that the intermediate sum exceeds the maximum value of the type (*e.g.*, the integer type) it will fail. For example, with $p = 6$ only 10 out of 120 orderings causes a failure. Doing tests with 10000 trial runs using send perturbation a failure was observed many times, in some cases within the first 128 runs. Contrast this result to the normal test runs where the failure was seldom observed and in some test runs there were no failures.

4.2 Race-addition problem

Artificially introducing a random delay in the sending of the messages will produce a race condition and the resulting

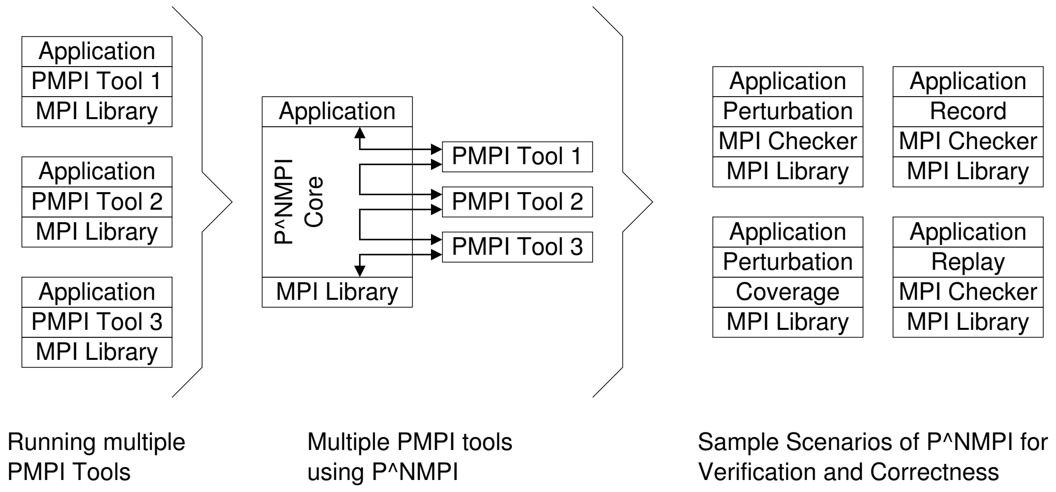

Figure 8: P^{N}MPI: Multiple concurrent PMPI Tools.

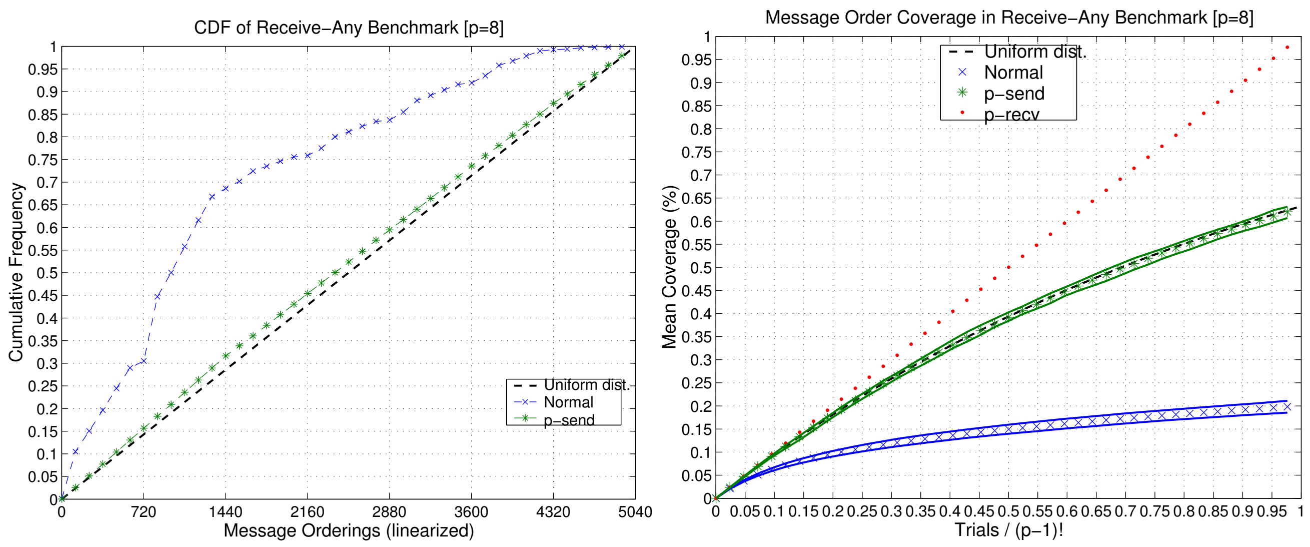

Figure 9: (Left) The empirical cumulative distribution functions over possible message orderings for the simple receive-any experiment in Figure 1. The unperturbed run differs significantly from a uniform distribution in which all message orderings are equally likely, while sender-side perturbation yields a good approximation. (Right) Coverage of possible message orderings in the receive-any experiment, where markers indicate mean values, and solid lines show minimum and maximum values in 100 trials. Simply randomly delaying sends improves coverage in this example by 3×, which nearly exactly matches the coverage if all orderings are equally likely.

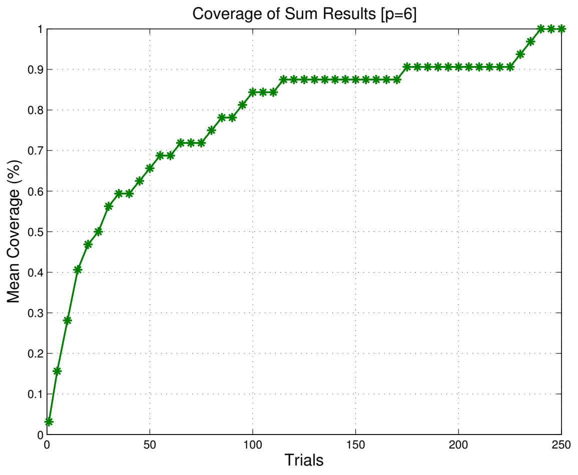

Figure 10: Resulting coverage of message orderings after n runs with sender-side perturbation of the race-addition problem of Figure 5.

value will be between '00000' and '11111', reducing the time-to-failure dramatically. Using the simple send-perturbation module with a maximum delay of $\frac{3}{10}$ seconds, we see all possible output values after 240 runs, with some combinations occurring more frequently than others as shown in Figure 10. We empirically selected the maximum delay time that produced all possible resulting values in the least number of trial runs.

4.3 Dining philosophers

We implemented the symmetric MPI dining philosophers protocol as described in Section 2.3. In this implementation, the user may specify that the time spent in the thinking and eating states be selected randomly from some distribution (*e.g.*, uniform or Gaussian) with a prescribed mean and standard deviation, μ_e and σ_e for the eating time, and μ_t and σ_t for the thinking time.

We evaluated the sender-based perturbation module of Section 3 that automatically selects the delay time, using a 4-process example where $\mu_e = \mu_t = 0.5$ sec, and $\sigma_e = \sigma_t = 0.1$ sec, for 1000 "meals" (*i.e.*, 1000 successful completions of think-eat iterations), both perturbed and unperturbed. To detect "failure," we allow each philsopher to time-out (starve).[2] The unperturbed runs never failed during these trials, since since the total time in non-acquisition states was 1.0 sec on average and the network latency relatively small (5 μs). By contrast, the perturbed example failed many times, with a mean-time-to-failure of 35.5 think-eat iterations, while incurring an overall execution time penalty of 1.5× (*i.e.*, in the perturbed case, the average think-eat iteration took 1.5 sec instead of 1 sec). Thus, we significantly improve the probability that the error occurs.

4.4 Convergence loop

We implemented the convergence loop model problem as described in Section 2.4, and ran a number of experiments at various values of p. We compare the empirically observed relative frequency of the bug occurring under no perturbation to sender- and receiver-side perturbation. For sender-side perturbation, we use a maximum delay time of 500μs.

For each p-process experiment, we use an automated search process to choose randomly all the individual elements of the sum and a suitable predicate, cond(s), so that the probability of failure, assuming all message orderings are equally likely, lies in some desired range. A failure occurs when two or more processors disagree about the result of the predicate. We fix the number of iterations of the outer-loop to be $2 \cdot p!$, instead of terminating the loop on cond(s) = false as shown in Figure 6. We then observe cond(s) to see when the processes disagree. Thus, each experiment produces a "failure rate" taken over the $2 \cdot p!$ trials. We repeated this experiment 30 times for each run mode, using the same initial data in each experiment. We present statistics on the failure rate under uncontrolled environmental noise conditions.[3]

The boxplots in Figure 11 show the distribution of the failure rate under each run mode, for $p = 4$ (left) and $p = 8$ (right). Higher failure rates mean a the bug is more likely to occur. We chose the initial data so that the theoretical failure rate is approximately 30.6% for $p = 4$ and 8.1% for $p = 8$, assuming all orderings are equally likely. The theoretical failure rate is the dashed horizontal line in each plot. The horizontal line in each box is the median failure rate in the 30 experiments, the box's vertical extents indicate lower and upper quartiles, and the whiskers show fences beyond which points (shown as asterisks) are outliers. The notches in each box show 95% confidence intervals about the median.

For both $p = 4$ and $p = 8$, the median failure rate for the normal mode is much lower than for p-send. Indeed, the normal median failure rate is 0: the bug never occurs at least half the time. Nonetheless, this mode exhibits the bug with high frequency for some experiments with $p = 4$ since there are inherently many such cases (30.6%) and any systematic/inherent bias toward certain orderings has a high probability of including one of the error cases. For $p = 8$, the inherent probability of detecting the error is much lower (8.1%). Thus, in only two instances (the outliers) did the failure rate exceed 4% without perturbation. By contrast, the send perturbations achieve the expected failure rates (as desired), with the expected rate at or just within the confidence interval.

The p-recv mode does as well as the p-send. Of course, the deterministic p-recv mode yields the same behavior on all runs. When $p = 8$, it does not achieve the naturally occurring rate. Our implementation is not "tuned" for this problem: the p-recv mode is independent but deterministic on all processes, and so the returned message orderings are correlated. For example, process 0 will always see the sequence $(1, 2, 3, 4, 5, 6, 7)$ when process 1 sees $(0, 2, 3, 4, 5, 6, 7)$ and 2 sees $(0, 1, 3, 4, 5, 6, 7)$, and so on. For practical bug detection support, the p-recv mode should implement the bookkeeping required to approximate a uniform distribution without replacement of possible message orderings.

5. EXTENSIONS AND OTHER TOOLS

We now explore the overall broader context of our research into perturbation and its combined use with other MPI test-

[2]Umpire [15] can now detect these deadlock conditions precisely if they are manifested in the actual message ordering.

[3]For example, we do not control what physical nodes are allocated to our job in this shared cluster, and cannot prevent network traffic from nearby nodes from influencing our experiments.

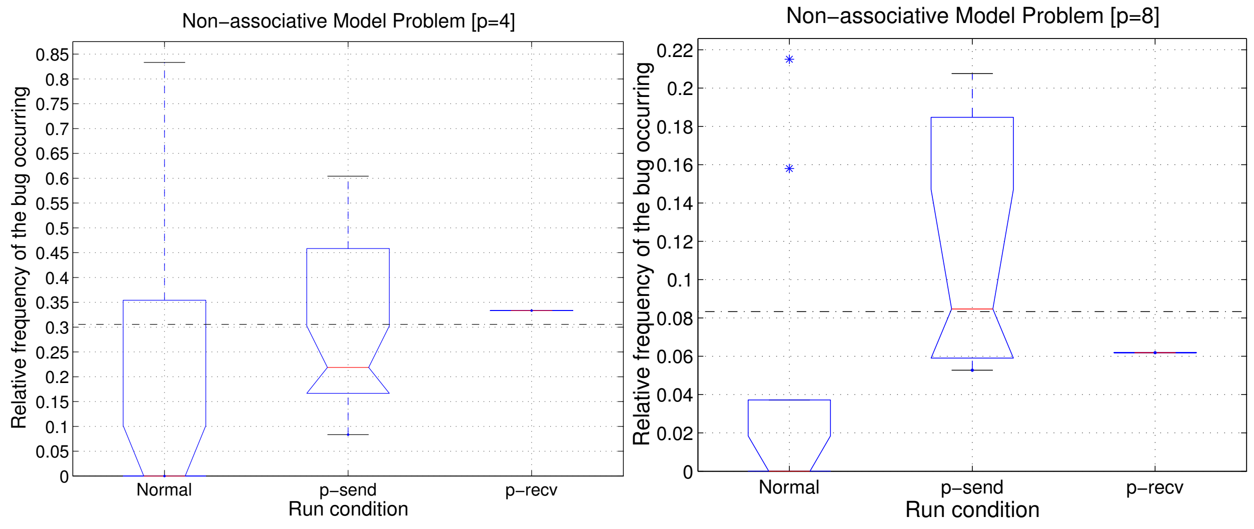

Figure 11: Examples of the frequency of bug occurrence in the convergence loop model problem under floating-point addition for $p = 4$ (left) and $p = 8$ (right). The dashed horizontal line in each plot is the theoretical probability of the bug, assuming all message orderings are equally likely.

ing tools. Specifically, we ask what other tools, techniques, and algorithms will improve testing and debugging of MPI programs. The P^{N}MPI infrastructure allows us to combine arbitrary and independent layers; we anticipate a variety of dynamic analysis and testing techniques, including capture layers in capture and replay-based test sessions, or layers that provide hooks to coverage analysis tools. We discuss these direct extensions of this work in Section 6. Here, we focus on useful tools for detecting bugs in MPI usage directly, using deep static analysis combined with dynamic analysis, and using the ROSE source-to-source compiler infrastructure as a basis for building such tools [13].

The MPI standard is a large and complex API that can easily be used incorrectly, particularly with the most recent additions of 1-sided communication and I/O in MPI-2 [12]. Static bug-pattern detectors, which require little or no program analysis could readily be applied to find simple MPI usage errors, such as incorrectly specifying the type of data passed into an MPI send or receive call. Extensions to many existing tools could support MPI specifically, including those proposed by Farchi, *et al.*, in the context of code reviews [6], and frameworks like FindBugs for general programs [7].

Although MPI-CHECK statically locates certain classes of MPI errors [10], we need deeper program analysis and detailed knowledge of the semantics of MPI to find many other kinds of MPI errors statically. For example, the control-flow of typical MPI programs depends on the unique rank of the process (line 4 of Figure 5 being a common pattern); this could be used to help match calls, such as sends and receives, barriers or other collectives. Conversely, we could find errors due to improper or non-existing call matchings. Dependence analysis could trace the flow of data that passes through MPI, and thereby check for common buffer errors in MPI programs, such as buffer overruns, reading from a receive buffer before a non-blocking receive completes, and using unitialized buffers, among others [3]. Other analysis and model checking approaches could be used to verify temporal usage properties (*e.g.*, non-blocking sends followed by waits), similar to recent work for I/O, operating system kernel, and threading library abstractions [5, 2].

Additional classes of tools are possible given the ability to instrument and transform the application in an automated fashion. For instance, consider that within a P^{N}MPI layer, each call only sees its context through the parameters passed to the MPI call. With a static analysis and instrumentation capability, it is possible to create callee-specific PMPI modules and combine them using the P^{N}MPI infrastructure into a single experiment. Such a capability would allow passing of additional, context-specific information to the tool layer to be used for testing.

6. CONCLUSIONS AND FUTURE WORK

The irritator techniques first used in IBM's ConTest tool for multithreaded programs apply naturally to MPI applications. We are actively identifying additional MPI usage patterns amenable to these techniques, extending the JitterBug implementation to be more complete (particularly with respect to receiver-side perturbations), and applying this work to applications beyond the model problems.

Though we argue perturbation is an important technique for improving the quality of tests, perturbation does not directly address the problem of automated detection. We view JitterBug as just one basic component in a more comprehensive MPI application testing package that includes integration with tools such as Umpire for MPI checking, coverage tools, and capture-replay tools, just as irritators are a component of ConTest. We are pursuing the combined use of these tools for MPI applications, using the P^{N}MPI as an essential framework for their use and development.

This paper considers what may be viewed a simple coverage model for MPI receive-any operations in which the goal is to realize all possible message orderings. We wish to develop true coverage models suitable for MPI, in the spirit of those developed for multithreaded applications [1].

7. REFERENCES

[1] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proc. Principles and Practice of Parallel Programming*, Chicago, IL, USA, June 2005.

[2] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proc. Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2004.

[3] J. DeSouza, B. Kuhn, and B. R. de Supinski. Automated, scalable debugging of MPI programs with the Intel Message Checker. In *Proc. 2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, MO, USA, May 2005.

[4] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3–5):485–499, 2003.

[5] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc.International Conference on Verification, Model Checking, and Abstract Interpretation*, Venice, Italy, 2004.

[6] E. Farchi and B. R. Harrington. Assisting the code review process using simple pattern recognition. In *Proc. IBM Verification Conference*, Haifa, Israel, November 2005.

[7] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices (Proceedings of Onward! at OOPSLA 2004)*, December 2004.

[8] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *Proc. Parallel Computing: Software Technology, Algorithms, Architectures, and Applications*, pages 493–500. Elsevier, 2004.

[9] D. Kranzlmüller and M. Schulz. Notes on non-determinism in message passing programs. In *Proc. 9th European PVM/MPI Users Group Meeting*, volume LNCS 2474, pages 357–367, Linz, Austria, October 2002. Springer.

[10] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.

[11] Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. http://www.mpi-forum.org/.

[12] Message Passing Interface Forum (MPIF). MPI-2: Extensions to the Message Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[13] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

[14] M. Schulz and B. R. de Supinski. A flexible and dynamic infrastructure for MPI tool interoperability. In *Proc. International Conference on Parallel Processing*, Columbus, Ohio, August 2006. (*to appear*).

[15] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proc. Supercomputing*. ACM/IEEE, 2000.