A Scalable and Distributed Dynamic Formal Verifier for MPI Programs

Anh Vo,
Sriram Aananthakrishnan,
and Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, Utah 84112-9205
Email: {avo,sriram,ganesh}@cs.utah.edu*

Bronis R. de Supinski,
Martin Schulz,
and Greg Bronevetsky
Center for Applied Scientific Computing
Lawrence Livermore National Lab
Livermore, California 96678-2391
Email: {bronis,schulzm,bronevetsky1}@llnl.gov*

***Abstract*—Standard testing methods of MPI programs do not guarantee coverage of all non-deterministic interactions (*e.g.*, wildcard-receives). Programs tested by these methods can have untested paths (bugs) that may become manifest unexpectedly. Previous formal dynamic verifiers cover the space of non-determinism but do not scale, even for small applications. We present DAMPI, the first dynamic analyzer for MPI programs that guarantees scalable coverage of the space of non-determinism through a decentralized algorithm based on Lamport-clocks. DAMPI computes alternative non-deterministic matches and enforces them in subsequent program replays. To avoid interleaving explosion, DAMPI employs heuristics to focus coverage to regions of interest. We show that DAMPI can detect deadlocks and resource-leaks in real applications. Our results on a wide range of applications using over a thousand processes, which is an order of magnitude larger than any previously reported results for MPI dynamic verification tools, demonstrate that DAMPI provides scalable, user-configurable testing coverage.**

I. Introduction

Almost all high-performance computing applications are written in MPI, which will continue to be the case for at least the next several years. Given the huge (and growing) importance of MPI, and the size and sophistication of MPI codes, scalable and incisive MPI debugging tools are essential. Existing MPI debugging tools have, despite their strengths, many glaring deficiencies. While existing tools include many detailed debugging and stack viewing features, they often fail to provide the needed insight into the error's root cause.

Errors in parallel programs are often non-deterministic, arising only infrequently (e.g., *Heisenbugs* [1]). MPI semantics encourage the creation of these errors through features such as `MPI_ANY_SOURCE` (wildcard, or non-deterministic) receives and non-deterministic probes. Existing tools and testing methodologies often provide little assistance in locating these errors [2]. Further, a given MPI implementation on any particular system tends to bias execution to towards the same outcomes for non-detemrinistic operations, which can mask errors related to them. Thus, these errors often only become manifest after moving to a new system or MPI implementation or even after making an apparently unrelated change to the source code. While random delays/sleeps inserted along computational paths can help improve fairness [3], these delays primarily modulate the time of that MPI calls are *issued* and provide no guarantees of increased coverage of the possible non-deterministic outcomes.

Our previous work achieves non-deterministic behavior coverage through an *active testing* or *dynamic formal verification* tool called ISP [4], [5], [6], [7], [8]. ISP uses the MPI profiling interface [9] to intercept MPI operations and to enforce particular outcomes for non-deterministic operations. In particular, the ISP scheduler employs an MPI-semantics aware algorithm that reorders or rewrites MPI operations before sending them into the MPI runtime. Thus, ISP can discover the set S of all sends that can match a non-deterministic receive. ISP then *determinizes* the MPI receives with respect to each $s \in S$, and then issues the MPI send and the determinized MPI receive into the MPI runtime so that they must be matched. Thus, ISP fully explores the range of non-deterministic outcomes for a given input, although the control flow decisions related to that input still limit overall testing coverage.

Using ISP, we successfully verified many MPI applications of up to 14K LOC for a few dozen processes, which is often sufficient to locate Heisenbugs. However, ISP's centralized scheduling algorithm is non-scalable and applying it to significantly larger process counts is infeasible. Further, ISP must delay non-deterministic outcomes even at small scales, which leads to long testing times. In effect, its scheduler poorly exploits the parallelism offered by the cluster on which the MPI program is being dynamically verified.

Not only do we need faster dynamic verification tools for modest scales, but many reasons motivate scalable tools:

- MPI programs often require at least some scale in order to run certain inputs due to memory sizes and other limits; how to modify the inputs for smaller scale runs is at best unclear (*e.g.*, if M is reduced, should N be reduced logarithmically?);
- Some bugs are only manifest *when* a problem is run

*This work was partially performed under the auspices of the U.S. Deparment of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. (LLNL-CONF-442475). The work of the Utah authors was partially supported by Microsoft.

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

at scale: some buffer overflows or some index exceeds a memory range; testing with equivalent smaller scale inputs cannot locate these errors;
- While user-level dynamic verification resolves significant non-determinism, testing at smaller scales may mask similar system-level errors, such as bugs in the MPI implementation.

We make the following contributions in this paper:

- The first dynamic MPI verifier to scale meaningfully: users can verify MPI codes within the parallel environment in which they develop and optimize them;
- A scalable algorithm based on *Lamport clocks* that captures possible non-deterministic matches;
- A characterization of the additional coverage of MPI patterns that rarely occur that we could achieve with a less scalable algorithm based on *vector clocks*;
- A *loop iteration abstraction* heuristic that allows programmers to indicate to the DAMPI scheduler which MPI loops contain non-deterministic operations;
- Another powerful heuristic, *bounded mixing*, that exploits our intuition that the effect of each non-deterministic MPI operation does not last long along the code path by using a series of overlapping finite height windows;
- Extensive demonstration of the DAMPI algorithm's guaranteed coverage on medium to large benchmarks.

Our loop iteration abstraction heuristic prevents naïve exploration of loops that can cause an unsustainably large number of paths through various non-deterministic choices. Instead, we allow the programmer to focus testing costs. Bounded mixing covers the whole MPI execution from `MPI_Initialize` to `MPI_Finalize`. We allow interactions only between non-deterministic commands within the same window. Thus, DAMPI can explore the MPI application state space over a summation of small exponentials of paths – and not all paths in the application which is an unimaginably large exponential.

Our benchmarks include the NAS Parallel Benchmarks (Fortran) and SpecMPI2007 (C/C++/Fortran), many of which contain a high degree of non-determinism. We also provide results for a new work-sharing library called ADLB [10]. This library from Argonne is loosely coupled, and aggressively employs non-deterministic commands. DAMPI is essential for ADLB, for which its non-deterministic commands are very difficult to control through all possible outcomes during conventional testing. Overall, DAMPI substantially increases the scale and applications to which we can practically apply dynamic MPI verification.

After providing background on Lamport clocks and the rare MPI patterns that they miss, we present the DAMPI algorithm in Section II, and results on a collection of real MPI parallel programming patterns in Section III. Section III-B. discusses our new complexity bounding approaches in DAMPI along with experiments.

II. Distributed Analyzer for MPI (DAMPI)

A. Background: Verifying MPI Programs with ISP

At a high level, ISP works by intercepting the MPI calls made by the target program and making decisions on when to send these MPI calls to the MPI library. It has two main components: the Interposition Layer and the Scheduler.

The Interposition Layer: The interception of MPI calls is accomplished by compiling the ISP interposition layer together with the target program source code. The interposition layer makes use of the MPI profiling interface. It provides its own version of *MPI_f* for each corresponding MPI function f. Within each of these MPI_f, the profiler communicates with the scheduler using Unix sockets to send information about the MPI call that the process is about to make. It then waits for the scheduler to decide whether to send the MPI call to the MPI library or to postpone it until later. When the permission to fire f is granted from the scheduler, the interposition layer issues the corresponding *PMPI_f* to the MPI run-time.

The ISP Scheduler: The ISP scheduler executes the central algorithm of ISP, which detects and enforces different outcomes of non-deterministic MPI events. The scheduler detects possible outcomes of non-deterministic events by fine-grain interaction with the MPI processes. Specifically, each MPI call involves a synchronous communication between the MPI process and the scheduler to enable the scheduler to impose a different issuing order of MPI calls (while still respecting MPI semantics). Thus, in effect, the scheduler can discover all possible outcomes resulting from one particular program run. The synchronous communication and the interception layer then dynamically convert each non-deterministic MPI call into its deterministic equivalents. For example, if the scheduler determines that an `MPI_Recv(MPI_ANY_SOURCE` can match sends from both P1 and P2 then it instructs the MPI process to issue the call into the MPI runtime as `MPI_Recv(0)` during the first interleaving, and `MPI_Recv(1)` during the second.

While the centralized scheduler easily maintains a complete global picture that facilitates the state space discovery process, it limits scalability. When the number of MPI calls become sufficiently large, the synchronous communication between the scheduler and the MPI processes becomes an obvious bottleneck. An early experimental version of ISP was developed in which MPI processes would be launched on different hosts and communicate with the scheduler through TCP sockets. This architecture removes the resource constraints faced by launching all MPI processes within one single machine. Nonetheless, all processes still synchronously communicate with the centralized scheduler. Figure 5 presents ISP overhead verifying ParMETIS [11] for different number of processes. As shown by the figure, the verification time under ISP quickly increases as the number of processes become larger which makes it infeasible to apply ISP to verification larger than a few hundred processes.

B. DAMPI: Rethinking ISP for the Distributed Setting

The key insight that allows us to design DAMPI's decentralized scheduling algorithm is that each non-deterministic

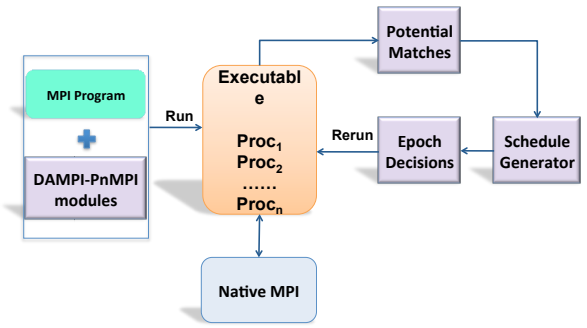

Fig. 1. DAMPI Framework

(ND) operation such as `MPI_Irecv(MPI_ANY_SOURCE)` or `MPI_Iprobe(MPI_ANY_SOURCE)` represents a point on the timeline of the issuing process when it *commits* to a match decision. We can naturally think of each such event as starting an *epoch* – an interval stretching from the current ND event up to (but not including) the next ND event. All deterministic receives can be assigned the same epoch within which they occur. Even though the epoch is defined by one ND receive matching another process's send, we can determine all *other* sends that can match it by considering all possible sends that are *not causally after* the ND receive (and subject to MPI's non-overtaking rules). We determine these sends using Lamport clocks [12] Based on these ideas, DAMPI's decentralized and scalable scheduling algorithm (demonstrated for a thousand processes) works on unmodified C/Fortran MPI programs as follows (also see Figure 1 and § II-E):

- All processes maintain "time" or "causality" through *Lamport clocks* (which are a frequently used optimization in lieu of the more precise but expensive Vector Clocks (we explain Lamport clocks in § II-C).
- Each MPI call is trapped (using P^{N}MPI instrumentation) at the node at which the call originates. For deterministic receive operations, the local process updates its own Lamport clock; for deterministic sends, it sends the latest Lamport clock along with the message payload using *piggyback messages*. All Lamport clock exchanges occur through piggyback messages.
- Each *non-deterministic* receive advances the Lamport clock of the local process. During execution, this receive matches one of the MPI sends targeting this process (or else we detect a deadlock and report it). However, each send that does not match this receive but impinges on the issuing process is analyzed to see if it is *causally concurrent* (computed using Lamport clocks). If so, we record it as a *Potential Match* in a file.
- At the end of the initial execution, DAMPI's scheduler computes the *Epoch Decisions* file that has the information to force alternate matches. DAMPI's scheduler then implements a depth-first walk over all Epoch Decisions (successively force alternate matches at the last step; then at the penultimate step; and so on until all Epoch Decisions are exhausted).

C. Background: Vector Clocks and Lamport Clocks

In the vector clock approach to keeping logical time, the current time of each process i in an N process system is maintained as an N-vector $VC[i]$. The jth component of $VC[i]$, denoted $VC_j[i]$, is process i's knowledge of j's time. Initially, all $VC[i]$s are zero vectors. Whenever process i performs a visible operation, its local time increases – *i.e.*, $VC_i[i]$ is incremented. Whenever process j sends a message to process i, it ships the current $VC[j]$ along with the message. Whenever process i receives a message from process j, $VC_k[i]$ is set to the maximum of $VC_k[j]$ and $VC_k[i]$ for every $k \in \{1 \ldots N\}$. Clock vectors are compared component-wise. If for all k $C_k[i] < C_k[j]$ then we say $C[i] < C[j]$, and in this case the event associated with $C[i]$ is before that associated with $C[j]$. Incomparable vector clocks (where $<$ does not hold either way) represent concurrent events.

Vector clocks are not scalable. Many researchers, such as Flanagan and Freund [13], have investigated scalable optimizations. However, these optimization tend to exploit domain knowledge. A common approach to scalability (at the expense of loss of precision) is to use *Lamport clocks*. We can think of Lamport clocks as a single integer approximating a vector clock, with similar update rules. If we maintain time using both vector and lamport clocks and we denote the Lamport clock of process i (j) by LC_i (LC_j), then $C[i] < C[j]$ implies $LC_i < LC_j$. However, $LC_i < LC_j$ does not imply $C[i] < C[j]$ – that is, Lamport clocks may order concurrent events.

Given all this, if a non-deterministic receive is associated with Lamport clock value b whereas an incoming send from process j impinges on process i with a Lamport clock a, we can say that the event associated with a is *not* causally after that associated with b (so a is causally before or concurrent with b). We call these sends *late arriving* (*late*). DAMPI's approach is to consider the *earliest* late send from each process as the potential alternate matches. This choice enforces MPI's message non-overtaking rule that requires messages sent between two processes with the same communicator and tag to arrive in the same order [14], as Figure 2 illustrates. The figure shows a wildcard receive event e in process R that has already been matched. The red arrows are the late messages with respect to e – hence, potential matches. Piggyback messages flow in at different real times (as shown by the dotted edges). The curved line termed the *causal line* helps visualize late messages. The causal line defines a frontier of events such that any event before it is not causally after any event on or subsequent to it.

In § II-E, we present the DAMPI algorithm in in its entirety and sketch its correctness. Correctness consists of two parts: (i) Soundness: *it will not find any ineligible matches*, and (ii) Completeness: that *it will find all potential matches*. We discuss a rare class of patterns for which completeness is not obtained in § V.

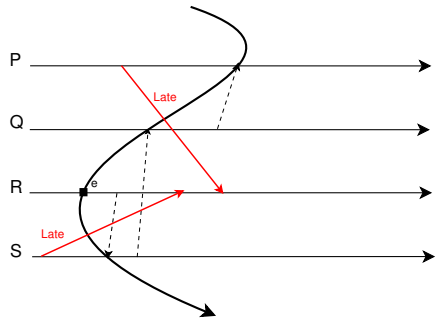

Fig. 2. Causal Line and Late Messages

D. Piggyback Messages

Piggyback data is sent along with regular messages to convey auxiliary information to the receiving process. In DAMPI, the piggyback data is the Lamport clock value of the process. The receiving process must match the piggyback data correctly with the associated regular message. We can use several mechanisms to communicate piggyback data including data payload packing, datatype packing, or the use of separate messages [15]. To ensure simplicity of implementation without sacrificing performance, DAMPI uses the separate message piggyback mechanism. Under this scheme, everytime a process sends a message m, a piggyback message m_p is sent (either before or after m); similarly, a process preparing to receive m must also post a receive for m_p. To ensure that the receiving processes can correctly associate m and m_p, DAMPI creates a shadow piggyback communicator for each existing communicator in the MPI program.

Receiving Wildcard Piggybacks: Receiving the piggyback for a wildcard receive is more complicated than for a deterministic receive, particularly for a non-blocking receive (`MPI_Irecv`), since the source is not known when the wildcard receive m is posted. We would often receive the wrong piggyback message (resulting in a tool-induced deadlock) if we blindly posted a wildcard receive to receive m_p subsequent to posting m. The DAMPI piggyback module addresses this problem by only posting the receive call for m_p after the completion of m (i.e., when `MPI_Wait` or `MPI_Test` is posted for the original `MPI_Irecv`). At that point, since we know the source of m, we can deterministically post the receive for m_p, thus ensuring receipt of the correct piggyback.

E. DAMPI Algorithm

The DAMPI algorithm (**Algorithm 1**) describes what each process P_i does in response to each message type m. We only show the pseudocode for `MPI_Irecv`, `MPI_Isend`, and `MPI_Wait` since they are the best candidates to represent the key ideas of the algorithm. We briefly discuss other MPI operations, including MPI collectives, `MPI_Test`, `MPI_Probe` and their variants, at the end of this section.

Initially, all processes automatically run through the whole program in "self-discovery" ($SELF\_RUN$) mode; that is, we allow the MPI runtime to determine the *first* matching

Algorithm 1 Pseudocode of P_i handling a message m

MPI_Init(argc,argv):
  if $ExistSchedulerDecisionFile()$ **then**
    $mode \leftarrow GUIDED\_RUN$
    $importEpochDecision()$
  end if
MPI_Irecv(m,src,req,comm):
  if $LC_i > guided\_epoch$ **then**
    $mode \leftarrow SELF\_RUN$
  end if
  if $src = MPI\_ANY\_SOURCE$ **then**
    if $mode = GUIDED\_RUN$ **then**
      $PMPI\_Irecv(m, GetSrcFromEpoch(LC_i), req, comm)$
    else
      $PMPI\_Irecv(m, src, req, comm)$
      $RecordEpochData(LC_i, src, req, comm)$
    end if
    LC_i++
  else
    $PMPI\_Irecv(m, src, req, comm)$
    $CreatePBReq(req)$
  end if
  if $src \neq MPI\_ANY\_SOURCE$ **then**
    $PMPI\_Irecv(m.LC, src, GetPBReq(req), GetPBComm(comm))$
  end if
MPI_Isend(m,dest,req,comm):

  $PMPI\_Isend(m, dest, req, comm)$
  $CreatePBReq(req)$
  $PMPI\_Isend(LC_i, dest, GetPBReq(req), GetPBComm(comm))$
MPI_Wait(req,status):
  $PMPI\_Wait(req, status)$
  if $req.type = ISEND$ **then**
    $PMPI\_Wait(GetPBReq(req), MPI\_STATUS\_IGNORE)$
  else
    MPI_Status status2;
    if $req.src \neq MPI\_ANY\_SOURCE$ **then**
      $PMPI\_Wait(GetPBReq(req), status2)$
    else
      $PMPI\_Recv(m.LC, status.MPI\_SOURCE, GetPBComm(comm))$
    end if
    $LC_i = max(LC_i, m.LC)$
    if $req.LC > m.LC$ **then**
      $FindPotentialMatches(status, req.comm, req.LC, m.LC)$
    end if
  end if

send for each wildcard receive. We associate the current LC of P_i, namely LC_i, with each wildcard receive e that P_i encounters through the *RecordEpochData* routine. This information helps to associate possible late messages with e. LC_i is then incremented, thus associating each wildcard receive with a unique LC_i value.

Whenever process P_i receives a message m sent to it by some other process, P_i extracts the LC field of m from the piggyback message m_p and compares the Lamport clock in m_p to LC_i. Message m is classified as late if $m.LC$ is less than LC_i. In this case, m will be matched against existing local wildcard receives (whose clocks are greater than $m.LC$) to see if they can be potential matches (they are actually matched according to tag, communicator, and also based on the MPI non-overtaking semantics mentioned earlier).

During subsequent replays (detected by the processes through the presence of the Epoch Decisions file), the program is run under guided mode (GUIDED_RUN), in which all matching sends up until the LC value *guided_epoch* are forced to be as per the information in the Epoch Decisions file, which is read by the *GetSrcFromEpoch* routine. After crossing *guided_epoch*, the execution reverts back to the $SELF\_RUN$ mode, which allows the algorithm to retrace previous matching decisions up to the *guided_epoch* and discover new non-deterministic possibilities. Our example in Figure 10 will clarify these ideas further.

MPI_Iprobe and MPI_Probe: Probes present another source of non-determinism because they can also accept `MPI_ANY_SOURCE` as the source argument. Thus, we treat wildcard probes just like wildcard receives for the purposes of matching late messages. The only difference is that we do not receive the piggyback messages, since probes do not remove any message from the incoming message queues. We only record a non-blocking probe (`MPI_Iprobe`) if the MPI runtime sets its *flag* parameter to true, which indicates that an an incoming message is ready to be received.

MPI Collectives: For each MPI collective operation such as `MPI_Barrier`, `MPI_Allreduce`, and `MPI_Bcast`, we update the LC of each participating process based on the semantics of the operation. For example, at an `MPI_Allreduce`, all processes will perform an `MPI_Allreduce` with the reduce operation `MPI_MAX` on their Lamport clock values. This choice reflects that every process effectively receives from all processes in the communicator. In contrast, at an `MPI_Bcast`, all processes receive (and incorporate) the Lamport clock of the *root* of the broadcast. While MPI semantics require that all processes in the communicator participate in the collective call, it does not require their synchronous completion. Handling the collective in this fashion ensures that we cover the widest range of MPI collective behaviors that the MPI standard allows.

MPI_Test: Upon the completion of the communication requests (signaled through a boolean flag), we treat `MPI_Test` and its variants similarly to `MPI_Wait`

Illustrative Example: The example in Figure 3 illustrates how DAMPI discovers errors through replay. In the initial execution

P0	P1	P2
Isend(to:1,22)	Irecv(*,x) if(x==33) error	Isend(to:1,33)

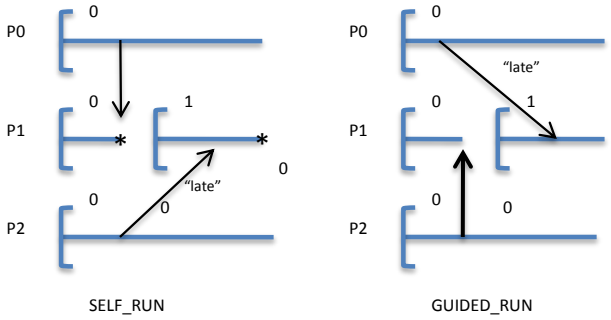

Fig. 3. Simple Example Illustrating How DAMPI Works

P0	P1	P2	P3
Isend(to:1)	Irecv(*)	Irecv(*)	Isend(to:2)
...	...	...	...
	Isend(to:2)	Isend(to:1)	

Fig. 4. Simple Example Illustrating Incompleteness Due to Lamport Clocks

in the *SELF_RUN* model, P0's send matches P1's wildcard receive, with P2's send coming in late. With respect to this log file, DAMPI does a depth-first unwinding of alternate matches. Then in the *GUIDED_RUN* mode, the resulting epoch decisions file forces P2 and P1 to match (shown by the heavy arrow), and the error is caught.

Correctness: The soundness of this algorithm follows from the background discussion of Lamport clocks. In particular, the *FindPotentialMatches* function is invoked under the correct circumstances when the Lamport clocks guarantee that it is a potential match (the send is not causally after the receive). The algorithm is also complete except for imprecision due to Lamport clocks, as we discuss next.

F. Imprecision of Lamport Clocks

Figure 4 discusses the situation in which Lamport clocks are not precise enough to guarantee completeness. In this figure, four processes are "cross coupled" in terms of non-determinism. Specifically, initially, both the wildcard receives (Irecv(*)) in P1 and P2 find only one matching send each (the first can match the send from P0 while the second can match the send from P3). Suppose we proceed with the P0 to P1 match. We then reach a state of execution in which P1's Isend(to:2) and P3's Isend(to:2) both are potential matches for P2's Irecv(*) but we do not detect the potential match of P2's Isend(to:1) for P1's Irecv(*). However if we initially proceeded with the P2 to P3 match, we would discover the full set of matches for P1's wildcard receive.

With vector clocks, DAMPI would work as follows:

- In the initial execution, we would proceed with the P0/P1 match and the P2/P3 match.
- Continuing along, P1's send would impinge on P2's timeline, and P2's send would impinge on P1's timeline. Because of the extra precision of vector clocks, both messages would be regarded as late. In effect, the Irecv(*)s maintain *incomparable* (concurrent) epoch values.

We cannot detect that the Irecv(*)s matched concurrent messages with Lamport clocks. Consequently, we record either P1's Isend(to:2) or P2's Isend(to:1) as not "late." In other words, we must judge one of these sends as *causally after* the Irecv(*) with which it could have matched. Vector clocks would provide completeness at the cost of scalability.

In our experiments with medium to large benchmarks, we have not encountered any other pattern where Lamport clocks lose precision other than indicated in this example. In our opinion, it is not worth switching to VCs just for the sake of these rare patterns. Static analysis may be able to detect MPI codes that have this pattern.

III. Experimental Results

We compare the performance of DAMPI and ISP. We also analyze it in terms of the state space reduction heuristics mentioned earlier. Our evaluation uses these benchmarks:

- An MPI matrix multiplation implementation, *matmult*;
- ParMETIS-3.1 [11], a fully deterministic MPI based hypergraph partition library;
- Several benchmarks from the NAS Parallel Benchmarks (NAS-PB) 3. [16] and SpecMPI2007 [17] suites;
- Adaptive Dynamic Load Balancing (ADLB) library [10].

Our ParMETIS, NAS-PB and SpecMPI tests measure DAMPI's overheads and target evaluation of its local error (e.g., request leak or communicator leak) checking capabilities. In *matmul*, we use a master-slave algorithm to compute $A \times B$. The master broadcasts the B matrix to all slaves and then divides up the rows of A into equal ranges and sends one to each slave. The master then waits (using a wildcard receive) for a slave to finish the computation. It then sends it another range r_j. This benchmark allows us to study the bounded mixing heuristic in detail with a well-known example. We also evaluate the bounded mixing heuristic with ADLB, a relatively new load balancing library that has significant non-determinism and an aggressively optimized implementation. In our previous experiments using ISP, we could not handle ADLB even for the simplest of verification examples. We now discuss our results under various categories.

A. Full Coverage

Figure 5 shows the superior performance of DAMPI compared to that of ISP running with Parmetis, which makes about one million MPI calls at 32 processes. As explained earlier, due to its centralized nature, ISP's performance quickly degrades as the number of MPI calls increases, while DAMPI exhibits very low overhead. In fact, the overhead of DAMPI on ParMETIS is negligible until the number of processes become large (beyond 1K processes). In order to understand

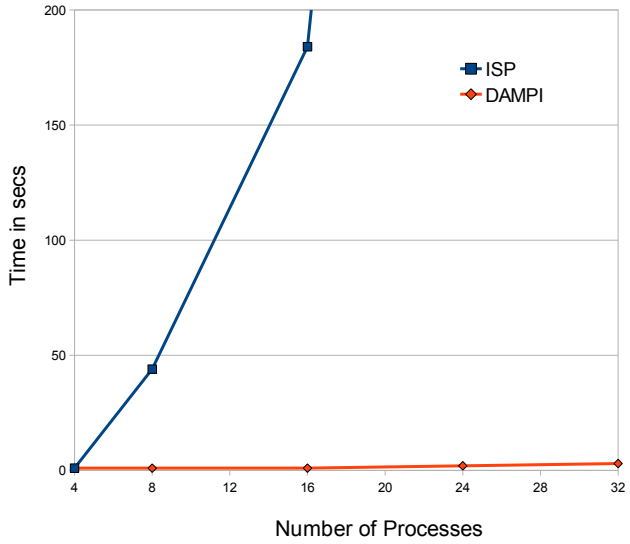

Fig. 5. ParMETIS-3.1: DAMPI vs. ISP

MPI Operation Type	*procs=8*	16	32	64	128
All	187K	534K	1315K	3133K	7986K
All per proc.	23K	33K	41K	49K	62K
Send-Recv	121K	381K	981K	2416K	6346K
Send-Recv per proc	15K	24K	31K	38K	50K
Collective	20K	36K	63K	105K	178K
Collective per proc	2.5K	2.2K	2.0K	1.6K	1.4K
Wait	47K	118K	272K	612K	1463K
Wait per proc	5.8K	7.3K	8.5K	9.6K	11K

TABLE I
Statistics of MPI Operations in ParMETIS-3.1

the reasons behind the significant improvement of DAMPI over ISP better, we log all MPI communication operations that ParMETIS makes (see Table I). We do not log local MPI operations such as `MPI_Type_create` or `MPI_Get_count`). We classify the operations as Send-Recv, Collective or Wait. Send-Recv includes all point-to-point MPI operations; Collective includes all collective operations; and Wait includes all variants of `MPI_Wait` (e.g., Waitall).

Although the total number of MPI operations grows by a factor of 2.5 on average as the number of the processes increases, the total number of MPI operations per process only grows by a factor of 1.3 on average. In effect, the number of MPI operations that the ISP scheduler must handle increases almost twice as fast as the number of MPI operations that each process in DAMPI nust handle as the number of processes increases (due to the DAMPI's distributed nature). Each type of MPI operation behaves similarly, especially the Collectives, for which the number of operations *per process* decreases as the number of processes increases. In addition to the increasing workload placed on the ISP scheduler, the large number of local MPI processes also stresses the system as a whole, which explains the switching from linear slowdown to exponential

Program	Slowdown	Total R*	C-Leak	R-Leak
ParMETIS-3.1	1.18x	0	Yes	No
104.milc	15x	51K	Yes	No
107.leslie3d	1.14x	0	No	No
113.GemsFDTD	1.13x	0	Yes	No
126.lammps	1.88x	0	No	No
130.socorro	1.25x	0	No	No
137.lu	1.04x	732	Yes	No
BT	1.28x	0	Yes	No
CG	1.09x	0	No	No
DT	1.01x	0	No	No
EP	1.02x	0	No	No
FT	1.01x	0	Yes	No
IS	1.09x	0	No	No
LU	2.22x	1K	No	No
MG	1.15x	0	No	No

TABLE II
DAMPI OVERHEAD: MEDIUM-LARGE BENCHMARKS AT 1K PROCS

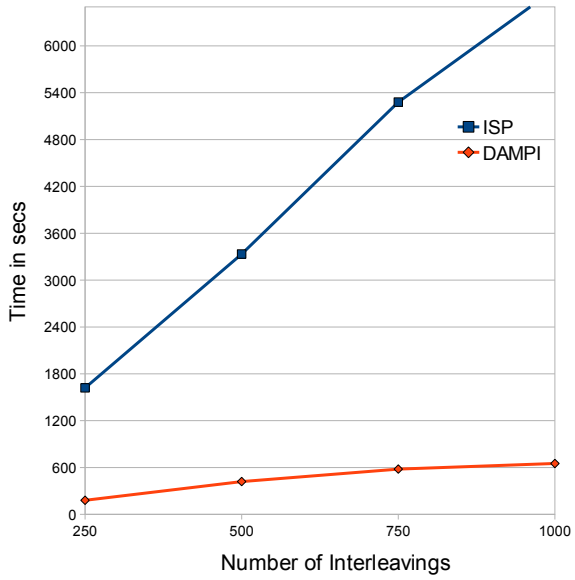

Fig. 6. Matrix Multiplication: DAMPI vs. ISP

slowdown around 32 processes. We also experimented with the version of ISP using TCP sockets but that version actually performed even worse compared to the normal ISP. These data further confirm our observation that the centralized scheduler is ISP's biggest performance bottleneck.

To evaluate the overhead of DAMPI further, we apply DAMPI on a range of medium to large benchmarks, including the NAS-NPB 3.3 suite and several codes from the SpecMPI2007 suite. We run the experiments on an 800 node, 16 core per node Opteron Linux cluster with an InfiniBand network running MVAPICH2[18]. Each node has 30GB of memory shared between all the cores. We submit all experimental runs through the Moab batch system and use the wall clock time as reported by Moab to evaluate the performance overhead. Table II shows the overhead of running DAMPI with 1024 processes. In Table II, the R* column gives the number of wildcard receives that DAMPI analyzed while C-leak and R-leak indicate if we detected any unfreed communicators and pending requests (not completed before the call to `MPI_Finalize`).

Figure 6 shows the time it takes for DAMPI and ISP to explore through the possible different interleavings of *matmul*. The experiments clearly show that DAMPI can offer coverage guarantees over the space of MPI non-determinism while maintaining vastly improved scalability when compared to ISP – the current state-of-the-art dynamic formal verifier for MPI programs. We attribute this improvement in handling interleavings to the lack of synchronous comminucation within DAMPI. All the extra communication introduced by DAMPI is done through MPI piggyback messages, which has been shown to have very low overhead [15]. However naïvely approaching the exponential space of interleavings in heavily non-deterministic programs is not a productive use of verification resources. We now present several heuristics implemented in DAMPI that can allow the user to *focus* coverage to particular regions of interest, often exponentially reducing the exploration state space.

B. Search Bounding Heuristics

Full coverage over the space of MPI non-determinism is often infeasible, even if desirable. Consider an MPI program that issues N wildcard receives in sequence, each with P potential matching senders. Covering this program's full state space would require a verifier to explore P^N inerleavings, which is impractical even for fairly small values (*e.g.*, $P = N = 1000$). While these interleavings represent unique message matching orders, most cover the same (equivalent) state space if the matching of one wildcard receive is independent of other matches. Consider these common communication patterns:

- A master/slave computation in which the master receives the computed work from the slaves and stores it in a vector indexed by the slave's rank;
- A series of computational phases in which processes use wildcard receives to exchange data and then synchronize.

Both patterns do not require that we explore the full state space. Clearly, the order of posting the master's receives does not affect the ending state of the program. Similarly, while the order of message matching within a single phase of the second pattern might lead to different code paths within a phase, the effect is usually limited to that particular phase.

Recognizing such patterns is a challenge for a dynamic verifier such as DAMPI, which has no knowledge of the source code. Further, complicated looping patterns often make it difficult to establish whether successive wildcard receives are issued from within a loop. Similarly, an `MPI_Allreduce` or an `MPI_Barrier` does not necessarily signal the end of a computation phase. Thus, it is valuable to capitalize on the knowledge of users who can specify regions on which to focus analysis. Such hints can significantly improve the coverage of *interesting* interleavings by a tool such as DAMPI. We now discuss our two *complementary* search bounding techniques,

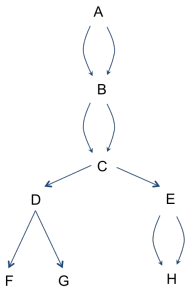

Fig. 7. A Simple Program Flow To Demonstrate Bounded Mixing

loop iteration abstraction and *bounded mixing search.*

1) Loop Iteration Abstraction: Many programs have loops with a fixed computation pattern that a verifier can safely ignore. By turning off interleaving exploration for non-deterministic matches occurring within such loops, DAMPI can explore other non-deterministic matches more thoroughly.

To use this feature in DAMPI, the user must insert `MPI_Pcontrol` calls at the beginning and end of loops that should not be explored. Upon logging these `MPI_Pcontrol` calls, DAMPI pursues only the matches it discovers during *SELF_RUN*, and avoids exploring alternative matches. Despite its simplicity, loop iteration abstraction can substantially reduce the iteration space that DAMPI must explore. In the future we will build static analysis based instrumentation facilities to semi-automate this heuristic.

2) Bounded Mixing: Many search bounding techniques exist. Bounded model checking [19] unravels the state space of a system to a finite depth. This heuristic suits hardware systems for which reachability graphs are considerably smaller than in software.

Context bounding [20] is much more practical in that it does not bias the search towards the beginning of state spaces. In effect, it runs a program under small *preemption quotas*. More specifically, special schedulers allow preemption two or three times *anywhere* in the execution. However, the scheduler can only employ a small fixed number of preemptions, after which it can switch processes only when they block.

While preemption bounding is powerful for shared memory concurrent programs based on threads, it is only marginally useful for message passing programs. In message passing, simple preemption of MPI processes is highly unlikely to expose new bugs (as explained earlier, one must take active control over their matchings). Also most preemptions of MPI programs prove useless since context-switching across deterministic MPI calls does not reduce the state space. We have invented bounded mixing, a new bounding technique that is tailor-made to how MPI programs work.

Intuition Behind Bounded Mixing: We have observed that each process of an MPI program goes through *zones* of computation. In each zone, the process exchanges messages with other processes and then finishes the zone with a collective operation (*e.g.* a reduction or barrier). Many such sequential zones cascade along – all starting from `MPI_Init` and ending in `MPI_Finalize`. In many MPI programs, these zones contain wildcard receives, and cascades of wildcard receives quickly end up defining large (exponential) state spaces. Figure 7 depicts an abstraction of this pattern. In this figure, A is a non-deterministic operation (e.g., a wildcard receive), followed by a zone followed by a collective operation. B then starts another zone and the pattern continues. If each zone contains non-deterministic operations then the possible interleavings is exponential in the number of zones (no interleaving explosion occurs if all zones contain only deterministic operations).

We observe that zones that are far apart usually do not interact much. We define the distance between two zones by the number of MPI operations between them. The intuition behind this statement is that each zone receives messages, responds, and moves along through a lossy operation (e.g., a reduction operation or a barrier). In particular, conditional statements coming later are not dependent on the computational results of zones occurring much earlier.

Based on these empirical observations above, bounded mixing limits the exploration of later zones to *representative paths* arriving at the zone instead of exploring *all paths* arriving at the zone. Thus, we explore the zones beginning at C only under the leftmost path A,B,C. We do not explore the zones beginning at C under all four paths. This example is actually bounded mixing with a mixing bound of $k = 2$ (the zones beginning at C and E are allowed to "mix" their states, and so do the zones beginning at C and D). We also allow the zones beginning at B and C to mix their states. Finally, we will allow the zones beginning at A and B to mix their states. Thus, bounded mixing is the "overlapping windows" analogy that we presented in § I.

Setting mixing bounds results in search complexity that grows *only as the sum of much smaller exponentials*. Using our example program with P^N possible interleavings earlier, a $k = 0$ setting will result in $P * N$ interleavings while a $k = unbounded$ setting will result in full exploration. Bounded mixing in DAMPI provides knobs that designers can set for various regions of the program: for some zones, they can select high k values while for others, they can select low values, which supports a selectively focused search.

Implementation of bounded mixing: We briefly explain how we implemented bounded mixing in DAMPI. Suppose the search is at some epoch s, and suppose s has several as yet unexplored potential matches *but all subsequent epochs of s have been explored*. Then, the standard algorithm will: (i) pursue the unexplored option at s, and (ii) recursively explore *all paths below that option*. In bounded mixing search, we will: (i) pursue the unexplored option at s, and (ii) recursively explore *all paths below that option* up to depth k. Thus, if B's right-hand side entry has not been explored, and if $k = 2$, then we will (i) descend via the right-hand side path out of B, and (ii) go only two steps further in all possible directions. After those k steps, we simply let the MPI runtime determine wildcard receive matching.

Experiments with bounded mixing: We first show the effects of bounded mixing on our small and simple application: *mat-*

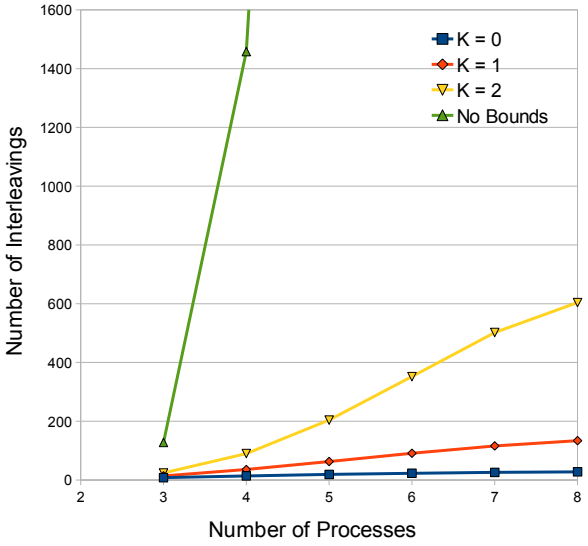

Fig. 8. Matrix Multiplication with Bounded Mixing Applied

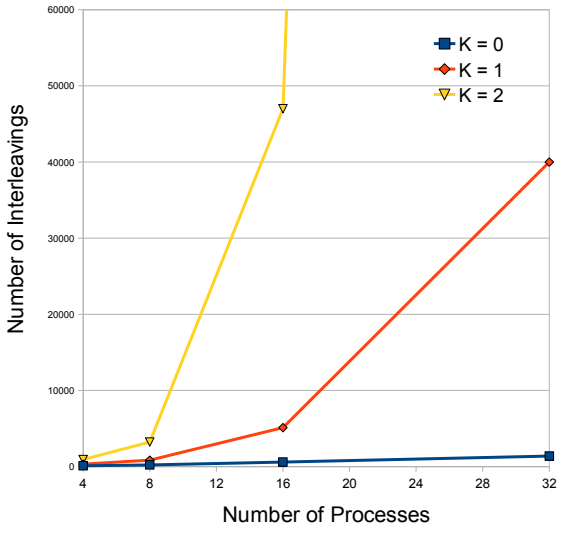

Fig. 9. ADLB with Bounded Mixing Applied

mul. Figure 8 shows the results of applying several different values of k. As expected, bounded mixing greatly reduces the number of interleavings that DAMPI explores. However, our heuristic has is another subtle yet powerful advantage: the number of interleavings increases in a linear fashion when k increases. Thus, users can slowly increase k should they suspect that the reaching effect of a matching receive is further than they initially assumed.

We also apply bounded mixing to the Asynchronous Dynamic Load Balancing (ADLB) library [10]. As the name suggests, ADLB is a highly configurable library that can run with a large number of processes. However, due to its highly dynamic nature, the degree of non-determinism of ADLB is usually far beyond that of a typical MPI program. In fact, verifying ADLB for a dozen processes is already impractical, let alone for the scale at which DAMPI targets. Figure 9 shows very encouraging results of verifying ADLB with various values of k

P0	P1	P2
Isend(to:1,22)	Irecv(*,x)	Barrier
Barrier	Barrier	Isend(to:1,33)
...	if(x==33) crash	...

Fig. 10. Example Illustrating DAMPI's Limitations

IV. Related Work

There are many conventional debugging tools for MPI programs. Tools such as TotalView [21] and STAT [22] do not help enhance non-determinism coverage; they are effective but only when an error actually occurs. Tools such as Marmot [23] and the Intel Message Checker [24] rely on schedule randomization, much like our discussions earlier pertaining to Jitterbug [3]. Another line of tools such as ScalaTrace [25] and MPIWiz [26] record MPI calls into a trace file and use this information to deterministically replay the program. However, these trace-based tools only replay the observed schedule. They do not have the ability to analyze the observed schedule and derive from them alternate schedules that can arise if non-deterministic matches are enforced differently. This crucial ability of DAMPI helps explore traces that may never natively appear on the given platform, but can suddenly show up (and potentially result in bugs) when the code is ported to another platform. In short, the aforementioned tools do not meet our stated objectives of guaranteed non-determinism coverage and scalability.

Among model checking tools, the only MPI model checker besides our own ISP tool is the MPI-SPIN tool [27], which requires users to hand-model their MPI codes in another notation (called Promela), which severely limits its usability. Our work on dynamic verification of MPI was inspired by Verisoft [28]. Also more recently, a dynamic verifier called CHESS [29] has been proposed for .NET codes. Our own group has developed a dynamic verifier called Inspect [30]. Verisoft, CHESS, and Inspect are tailored to verify shared memory thread programs. They do not have the instrumentation or controlled dynamic replay capabilities needed for MPI programs. Both ISP and DAMPI are unique in their class.

V. Limitations

One limitation of the DAMPI algorithm (beyond the imprecision caused by Lamport clocks) is illustrated in the example in Figure 10. In this example, it is possible to crash the program under some MPI runtimes because the `Barriers` can be crossed by merely issuing the `Isend` and `Irecv` from P0 and P1; this allows P2's `Isend` to be a competitor for P1's `Irecv`! The reason why we can't detect this failure in DAMPI is because upon `Irecv`, we are updating the

process local Lamport clock; and `Barrier` propagates this knowledge globally *even though* `Irecv` has not completed (its `Wait`/`Test` has not been encountered).

The omission pattern can be succintly stated as follows: if a non-deterministic `Irecv` is followed by any operation (`Barrier` or `Send`) that sends the updated clock value *before* a `Wait`/`Test` is seen, then the DAMPI algorithm is vulnerable. Fortunately we can check this pattern dynamically, and local to each process in a scalable manner. *We have implemented such a monitor in DAMPI and it has not failed in all our tests.* Thus, DAMPI is capable of alerting when users encounter this pattern. We are investigating more elaborate mechanisms to detect and to handle this pattern correctly (basically using a pair of Lamport clocks – one for handling wildcard receives, and the other for transmittal to other processes). These Lamport clocks will be synchronized when a `Wait`/`Test` is encountered.

VI. Concluding Remarks

Verifying MPI programs over the space of MPI non-determinism is a challenging problem, especially at large scales. Existing MPI tools either lack coverage guarantees or do not scale well. In this paper we present DAMPI, a scalable framework that offers coverage guarantees for programs that use non-deterministic MPI calls. Our contributions include a novel method for detecting different outcomes of a non-deterministic receive without relying on a centralized process/thread. These different outcomes can be enforced through replays. We also present two different search bounding heuristics that provide the user with the ability to limit the coverage to areas of interests. We report our results on applying our tools on medium to large benchmarks running with thousands of processes, many of which make extensive use of non-deterministic calls. DAMPI is the first (and currently only) tool that can guarantee coverage at such scales.

Future Work: We are working on additional heuristics and analyses that can further enhance the usability and scalability of the tool. One current topic of interest is to recognize patterns of MPI operations and to determine whether we can safely ignore such regions of code during testing. Our user-annotated loop coverage reduction strategy presented earlier is the first step in this area. An automatic detection mechanism will certainly make debugging MPI applications at large scales much easier.

References

[1] J. Gray, "Why do computers stop and what can be done about it?" in *Symposium on Reliability in Distributed Software and Database Systems (SRDS)*, 1986, pp. 3–12.

[2] http://www.cs.utah.edu/formal_verification/ISP_Tests/.

[3] R. Vuduc, M. Schulz, D. Quinlan, and B. R. de Supinski, "Improving distributed memory applications testing by message perturbation," in *Parallel and Distributed Systems: Testing and Debugging Workshop (PADTAD)*, 2006.

[4] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, "Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings," in *International Conference on Computer Aided Verification (CAV)*, 2008, pp. 66–79.

[5] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Implementing efficient dynamic formal verification methods for MPI programs," in *EuroPVM/MPI*, 2008.

[6] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur, "Formal verification of practical MPI programs," in *ACM Conference on Principles and Practices of Parallel Programming (PPoPP)*, 2009, pp. 261–269.

[7] A. Vo, S. S. Vakkalanka, J. Williams, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Sound and efficient dynamic verification of MPI programs with probe non-determinism," in *EuroPVM/MPI*, 2009, pp. 271–281.

[8] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby, "Reduced execution semantics of MPI: From theory to practice," in *International Symposium on Formal Methods (FM)*, 2009, pp. 724–740.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[10] R. Lusk, S. Pieper, R. Butler, and A. Chan, "Asynchronous dynamic load balancing," http://www.cs.mtsu.edu/ rbutler/adlb/.

[11] G. Karypis, "METIS and ParMETIS," http://glaros.dtc.umn.edu/gkhome/views/metis.

[12] L. Lamport, "Time, clocks and ordering of events in distributed systems," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[13] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 121–133.

[14] "MPI standard 2.1," http://www.mpi-forum.org/docs/docs.html.

[15] M. Schulz, G. Bronevetsky, and B. R. de Supinski, "On the performance of transparent MPI piggyback messages," in *EuroPVM/MPI*, 2008, pp. 194–201.

[16] http://www.nas.nasa.gov/Resources/Software/npb.html.

[17] http://www.spec.org/mpi.

[18] http://mvapich.cse.ohio-state.edu/.

[19] http://www.cprover.org/cbmc/.

[20] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 446–455, 2007.

[21] T. Software, http://www.etnus.com/Products/TotalView/.

[22] https://computing.llnl.gov/code/STAT/.

[23] B. Krammer, K. Bidmon, M. S. Mller, and M. M. Resch, "Marmot: An MPI analysis and checking tool," in *Parallel Computing 2003*, Sep. 2003.

[24] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of MPI programs with Intel®message checker," in *International Workshop on Software Engineering for High Performance Computing Applications (SE-HPCS)*, 2005, pp. 78–82.

[25] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696–710, 2009.

[26] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker, "MPIWiz: Subgroup reproducible replay of MPI applications," in *In Principles and Practice of Parallel Programming (PPoPP)*, 2009, pp. 251–260.

[27] S. F. Siegel and G. S. Avrunin, "Verification of MPI-based software for scientific computation," in *International SPIN Workshop on Model Checking Software (SPIN)*, 2004, pp. 286–303.

[28] P. Godefroid, B. Hanmer, and L. Jagadeesan, "Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor," *Bell Labs Technical Journal*, vol. 3, no. 2, April-June 1998.

[29] http://research.microsoft.com/en-us/projects/chess/.

[30] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Distributed dynamic partial order reduction based verification of threaded software," in *International SPIN Workshop on Model Checking Software (SPIN)*, 2007, pp. 58–75.