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, UT 84112
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 9678-2391
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 in the production use. Previous formal dynamic verifiers cover the space of non-determinism but do not scale beyond dozens of MPI processes even for small applications. We present DMA, the first dynamic formal analysis tool for MPI programs that guarantees scalable coverage of the space of MPI non-determinism through a decentralized algorithm based on Lamport Clocks. DMA computes alternative non-deterministic matches and then enforces them in subsequent replays of the MPI program. To avoid interleaving explosion, DMA employs heuristics allowing the programmer to focus coverage to regions of interest. We show that DMA can detect deadlocks and resource leaks in real applications. Our results on a wide range of applications using over a thousand MPI processes, which is an order of magnitude larger than any previously reported results for MPI dynamic verification tools, demonstrate that DMA 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). 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 [1]. Further, a given MPI implementation on any particular system tends to bias execution to towards the same outcomes for non-deterministic 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 [2], 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 [3], [4], [5], [6], [7]. ISP uses MPI profiling interface [8] to intercept MPI operations and to enforce particular outcomes for non-deterministic ones. 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 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 verifer to scale meaningfully: users can verify MPI codes within the parallel environment in which they develop and optimize them;
• A scalable Lamport clock based algorithm to capture 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 DMA 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 DMA 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 on 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, DMA can explore the state space of the MPI application 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 [9]. This library from Argonne is loosely coupled, and aggressively employs non-deterministic commands. DMA is essential for ADLB, for which its non-deterministic commands are very difficult to control through all possible outcomes during conventional testing. Overall, DMA 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 DMA 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 DMA along with experiments.

II. THE DISTRIBUTED MPI ANALYSIS (DMA)

a) DMA: Rethinking ISP for the Distributed Setting:
The key insight that allows us to design the decentralized scheduling algorithm of DMA is that non-deterministic (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. It is natural to 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 as they occur within. Even though the epoch is defined by one ND receive matching another process’s send, how can we determine all other sends that can match it? The solution is to pick all those sends that are not causally after the ND receive (and subject to MPI’s non-overtaking rules). We determine these sends using Lamport clocks [10] Based on these ideas, DMA’s decentralized and scalable scheduling algorithm (demonstrated for a thousand processes) works as follows (Figure 1 and § II-C) on unmodified C/Fortran MPI programs:

• 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 (Lamport clocks are explained in § II-A).
• Each MPI call is trapped (using P^N(MPI instrumentation) at the node where 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 piggy-back messages. All Lamport clock exchanges occur through piggy-back messages.
• Each non-deterministic receive advances the Lamport clock of the local process. During execution, this receive will have matched one of the MPI sends targeting this process (else we would have caught a deadlock and reported 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, it is recorded as a Potential Match in a file.
• At the end of the initial execution, DMA’s scheduler computes the Epoch Decisions file which has the information to force alternate matches. Now, DMA’s scheduler proceeds to carry out a depth-first walk over all Epoch
Decisions (replay alternative matches at the last step; then at the penultimate step; and so on till the Epoch Decisions are exhausted).

A. 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 $j$th component of $VC[i]$, denoted $VC[j][i]$, is process $i$’s knowledge of $j$’s time. Initially, all $VC[j][i]$ 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 $VC[i]$ 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 $k \in \{1 \ldots N\}$, $C_k[i] < C_k[j]$ 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.

It is well known that vector clocks are not scalable. There are many scalable optimizations that have been investigated by other researchers – e.g. [11] for a recent work. Each such optimization tends to exploit domain knowledge. One common approach to scalability (at the expense of loss of precision) is to use Lamport clocks. One can think of Lamport clocks as a single integer approximating a vector clock, with similar updation rules. If one were to maintain time using both vector and lamport clocks and we denote the lamport clock of process $i$ by $LC_i$, then it is known that $C[i] < C[j]$ implies $LC_i < LC_j$. It is also true that even if $C[i]$ and $C[j]$ are not comparable (concurrent events), $LC_i < LC_j$.

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). DMA’s approach is to consider the earliest late send from each process as the potential alternate matches. This choice is to enforce MPI’s message non-overtaking rule which says that messages sent between two processes with the same communicator and tag must arrive in the same order [12]. Figure 2 illustrates this situation. Here, we show a single wildcard receive event in process $R$ that has presumably found a match. The red arrows are the late messages with respect to $e$ – hence are potential matches. Piggyback messages flow in at different real-times (as shown by the dotted edges). The curved line termed 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-C, we present the DMA algorithm in all of 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. There are a rare class of patterns where completeness is not obtained, and this is discussed in § V.

B. Piggyback Messages

Piggyback data is sent along with regular messages to convey auxiliary information to the receiving process. In DMA, the piggyback data is the Lamport clock value of the process. It is crucial for the receiving process to match the piggyback data correctly with the associated regular message. Several mechanisms are available for communicating piggyback data including data payload packing, datatype packing, or the use of separate messages [13]. To ensure simplicity of implementation without sacrificing performance, DMA 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 $m$); likewise, 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 $m_p$, DMA creates a shadow piggyback communicator for each existing communicator in the MPI programs.

b) Receiving Wildcard Piggybacks:: Receiving piggyback messages corresponding to wildcard receives is not as straight-forward as deterministic receive, especially in the case of non-blocking receives (MPI_Irecv) since the source is not known at the time a wildcard receive $m$ is posted. It is possible (and often the case) to receive the wrong piggyback message (resulting in a tool-induced deadlock) if the system blindly posts a wildcard receive to receive $m_p$ subsequent to posting $m$. To address this problem, the DMA piggyback module will delay posting the receive call for $m_p$ until the completion of $m$ (i.e., when MPI_Wait or MPI_Test is posted for the original MPI_Irecv). At that point, since the source of $m$ is known, we can post the receive for $m_p$ as a deterministic receive, thus ensuring correct receipt of the piggyback message.

C. DMA Algorithm

The DMA 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. Other MPI calls including MPI
Algorithm 1 DMA Pseudocode for $P_i$ and incoming/outgoing message $m$

```plaintext
MPI_Init(argc,argv):
    if ExistSchedulerDecisionFile() then
        mode ← GUIDED_RUN
        importEpochDecision()
    end if

MPI_Irecv(m,src,req,comm):
    if LC_i > guided_epoch then
        mode ← 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 ≠ MPI_ANY_SOURCE then
        PMPI_Irecv(m,LC_i,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 ≠ MPI_ANY_SOURCE then
            PMPI_Wait(GetPBReq(req), status2)
        else
            PMPI_Recv(m,LC_i,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
```

Collective calls, MPI_Test, MPI_Probe and their variants are discussed briefly at the end of this section.

Initially, all processes automatically run through the whole program in "self-discovery" (SELF_RUN) mode; that is, the first matching sends for a wildcard receive is determined by the MPI runtime itself. For each wildcard receive $e$ that $P_i$ encounters, the current $LC$ of $P_i$, namely $LC_i$, is associated with the $e$ with the help of the call RecordEpochData. This procedure helps match up $e$ with possible late messages that may be encountered subsequently. $LC_i$ 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 determined 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. This is effected by the call GetSrcFromEpoch. After crossing guided_epoch, the execution reverts back to the SELF_RUN mode. This allows the algorithm to retrace the previous matching decisions up to the point guided_epoch and discover new non-deterministic possibilities. Our example in Figure 10 will clarify these ideas further.

1c) 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, wildcard probes are treated just like wildcard receives when it comes to matching late messages. The only difference is that they do not receive the piggyback messages, which reflects the fact that probes do not actually remove any message from the incoming message queues. MPI_Iprobe are only recorded when its flag option is set to true by the MPI runtime (signaling the presence of an incoming message ready to be received).

d) MPI Collectives: At MPI collective calls such as MPI_Barrier, MPI_Allreduce, and MPI_Bcast, the $LC$ of the participating processes will be updated depending on the nature of the collective call. 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 helps every process to “catch up” with the logical time of each other process. In contrast, at an MPI_Bcast, all processes will receive (and incorporate) the Lamport clock of the broadcast root of the broadcast. While the MPI semantics requires 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
A vector clock based implementation of DMA 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 P3’s send would impinge on P1’s timeline.
- Because of the extra precision possessed by vector clocks, both these would be regarded as late messages. In effect, both Irecv(*)’s can maintain incomparable (concurrent) epoch values.
- With Lamport clocks, the Irecv(*) cannot maintain an incomparable Lamport clock. Consequently, either P1’s Isend(to:2) or P2’s Isend(to:1) will end up being regarded as not “late.” In other words, one of these sends will be judged to be causally after the Irecv(*) with which it could have matched.

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 forewarn us of MPI codes that have this pattern.

### III. Experimental Results

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

- An MPI matrix multiplication implementation, matmult;
- ParMETIS-3.1 [14], a fully deterministic MPI based hypergraph partition library;
- Several benchmarks from the NAS Parallel Benchmarks (NAS-PB) 3. [15] and SpecMPI2007 [16] suites;
- Adaptive Dynamic Load Balancing (ADLB) library [9].

Our ParMETIS, NAS-PB and SpecMPI tests measure DMA’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 aggressively optimized in its 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 DMA compared to that of ISP running with ParMETIS, which makes about one million MPI calls at 32 processes. Due to its centralized nature, ISP’s performance quickly degrades as the number of MPI calls increases, while DMA exhibits very low overhead. In fact, the overhead of DMA is negligible until the number of processes become large (beyond 512/1K processes).

To further evaluate the overhead of DMA, we apply DMA 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 a 800 node, 16 cores per node Opteron Linux cluster with InfiniBand network running MVAPICH2[17]. 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 I shows the overhead of running DMA with 1024 processes. In Table I, the R* column gives the number of wildcard receives that DMA analyzed while C-leak and R-leak give the number of unfreed communicators and pending requests (not completed before the call to MPI_Finalize).

Figure 6 shows the time it takes for DMA and ISP to explore through the possible different interleavings of matmul. The experiments clearly show that DMA 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. 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 DMA 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 with each receive having $P$ potential matching senders. Covering this program’s full state space would require a verifier to explore is $P^N$ 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 the following common MPI 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Overhead</th>
<th>$R^*$</th>
<th>C-Leak</th>
<th>R-Leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParMETIS-3.1</td>
<td>1.18x</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>104.milc</td>
<td>15x</td>
<td>51K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>107.leslie3d</td>
<td>1.14x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>113.gemsfjtd</td>
<td>1.13x</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>126.lammps</td>
<td>1.88x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>130.socorro</td>
<td>1.25x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>131.lj</td>
<td>1.04x</td>
<td>732</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BT</td>
<td>1.28x</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CG</td>
<td>1.09x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DT</td>
<td>1.01x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>EP</td>
<td>1.02x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FT</td>
<td>1.01x</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IS</td>
<td>1.09x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LU</td>
<td>2.22x</td>
<td>1K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MG</td>
<td>1.15x</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

TABLE I
DMA OVERHEAD: MEDIUM-LARGE BENCHMARKS AT 1K PROCS

Fig. 6. Matrix Multiplication: DMA vs. ISP
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, the order of message matching within a single phase of the second pattern might lead to different code paths within a phase, the effect of such match is usually limited to that particular phase.

Recognizing such patterns is a challenge for a dynamic verifier such as DMA, 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 DMA. We now discuss our two complementary search bounding techniques, loop iteration abstraction and bounded mixing search.

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

To use this feature in DMA, the user must insert MPI_Pcontrol calls at the beginning and the end of loops they want to avoid exploring. Upon logging these MPI_Pcontrol calls, DMA 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 DMA 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 [18] 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 [19] 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, simply 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.

\[ P \]

\[ N \]

\[ k \]

\[ \text{Fig. 7. Illustration of MPI Program Flows} \]

\[ a) \text{Intuition Behind Bounded Mixing:} \] 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 (reduction, barrier, etc.). 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 extremely large (exponential) state spaces. Figure 7 depicts an abstraction of this pattern. In this figure, A is a non-deterministic receive (say), and there is a zone following that. Then there is a collective of some kind, followed by B which starts another zone, etc. What we depict is really a sequence of zones each containing non-deterministic operations (if all zones contained only deterministic operations, there would not be an interleaving explosion).

With such patterns, it is clear that we have an exponentially growing number of paths with the number of sequential zones. We observe that often zones far apart do not interact much (let us define the distance between two zones by the number of MPI calls between them). The intuition behind this statement is that each zone in a sense receives messages, responds, and moves along through a lossy operation (a reduction operation, a barrier, etc.). In particular, conditional statements coming later are not dependent on the computational results of zones occurring much earlier.

Here is the bounded mixing idea in a nutshell: based on our empirical observations above, we can limit the exploration of later zones under representative paths arriving at the zone – not under all paths arriving at the zone. Thus, we will explore the zones beginning at C only under the leftmost path A,B,C. We will not explore the zones beginning at C under all four of these paths. This 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 will 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. This is the “overlapping windows” analogy that we presented in § I.

The advantage of setting mixing bounds is of course that the search complexity would grow 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 \( k = \text{unbounded} \) setting will result in full exploration. What we really provide are 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. This allows the search to be selectively focused.

**Implementation of bounded mixing:** We briefly explain how bounded mixing search was implemented in DMA. Suppose the search is at some epoch \( s \), and suppose \( s \) has some several potential matches not yet explored 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 may simply let the MPI runtime take over the wildcard receives matching. This precisely realizes ‘windows’ alluded to above.

**Experiments with bounded mixing:** We first show the effects of bounded mixing on our small and simple application: *matmul*. Figure 8 shows the results of applying several different values of \( k \). As expected, the number of interleavings that DMA explores for *matmul* greatly reduces. However, there is another subtle yet powerful advantage of this heuristic: the number of interleavings increases in a linear fashion when \( k \) increases. This allows the users to slowly increase \( k \) should they suspect that the reaching effect of a matching receive is further than what was initially assumed.

We then applied bounded mixing to the Asynchronous Dynamic Load Balancing, ADLB [9]. As the name suggests, ADLB is a highly configurable library that can be setup to 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 dozens of processes is already impractical, let alone for the scale at which DMA targets. Figure 9 shows very encouraging results of verifying ADLB with various values of \( k \).

**IV. Related Work**

There are a large number of conventional debugging tools for MPI programs. Tools such as TotalView [20] and STAT [21] do not help enhance non-determinism coverage; their impressive help becomes available once an error has occurred. Tools such as Marmot [22] and the Intel Message Checker [23] rely on schedule randomization, much like our discussions earlier pertaining to Jitterbug [2]. These 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 [24] which requires users to hand-model their MPI codes in another notation (called Promela). This severely limits the usability of MPI-SPIN. Our work on dynamic verification of MPI was inspired by Verisoft [25]. Also more recently, a dynamic verifier called CHESS [26] has been proposed for .NET codes. Our own group has developed a dynamic verifier called Inspect [27]. 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 DMA are unique in their class.
V. LIMITATIONS

One limitation of the DMA 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 DMA 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 succinctly 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 DMA 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 DMA and it has not failed in all our tests. Thus, DMA is capable of alerting when users are running into this pattern. More elaborate mechanisms (to detect and correctly handle this pattern) are under investigation (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 nondeterminism is a challenging problem, especially at large scale. The existing MPI tools either lack coverage guarantee or do not scale well. In this paper we present DMA, a scalable framework that offer coverage guarantee for programs that use nondeterministic 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. To the best of our knowledge, DMA is the first and only tool that can guarantee coverage at such scale. Future Work: We are working on even more heuristics and analysis that can further enhance the usability and scalability of the tool. One current topic of interest is to recognize the patterns of MPI calls and determine whether such regions of code can be safely ignored 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 scale much easier.

REFERENCES