Large Scale Verification of MPI Programs
Using Lamport Clocks with Lazy Update

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

Bronis R. de Supinski,
Martin Schulz,
and Greg Bronevetsky
Center for Applied Scientific Computing
Lawrence Livermore National Lab
Livermore, California 96687-2391
Email: {bronis,schulzm,bronevetsky}@llnl.gov*

Abstract—We propose a dynamic verification approach for large-scale message passing programs to locate correctness bugs caused by unforeseen nondeterministic interactions. This approach hinges on an efficient protocol to track the causality between nondeterministic message receive operations and potentially matching send operations. We show that causality tracking protocols that rely solely on logical clocks fail to capture all nuances of MPI program behavior, including the variety of ways in which nonblocking calls can complete. Our approach is hinged on formally defining the matches-before relation underlying the MPI standard, and devising lazy update dynamic logical clock based algorithms that can correctly discover all potential outcomes of nondeterministic receives in practice, can achieve the same coverage as a vector clock based algorithm while maintaining good scalability. LLCP allows us to analyze realistic MPI programs involving a thousand MPI processes, incurring only modest overheads in terms of communication bandwidth, latency, and memory consumption.

I. INTRODUCTION

MPI (Message Passing Interface, [7]) continues to be the most widely used API for writing parallel programs that run on large clusters. MPI allows programmers to employ many high performance communication constructs such as asynchronous (nonblocking) and synchronous communication calls, nondeterministic constructs, and persistent communication requests. Its feature-rich semantics provide performance and flexibility, but also create several debugging challenges.

Buggy MPI programs, especially under the presence of nondeterminism, are notoriously hard to debug. Bugs may lurk in nondeterministic execution paths that are not taken on a given testing platform. The key problem we solve in this paper is: how does one affordably detect missed nondeterministic choices and force executions to occur along them? To illustrate how difficult this problem becomes with MPI, we consider the program in Figure 1 in which an asynchronous nondeterministic receive posted in process $P_1$ can potentially match with messages sent by either $P_0$ and $P_2$. Under traditional testing, one may never successfully be able to force $P_2$’s message (which triggers ERROR) to match. While this option appears impossible due to its issuance after an MPI barrier, it is indeed a possible match because MPI semantics allow a nonblocking call to pend until its corresponding wait is posted. This example illustrates the need for more powerful verification techniques than ordinary random testing on a cluster in which $P_2$’s match may never manifest itself due to the absolute delays and, yet, it may show up when the code is ported to a different machine.

In this paper, we show how one can achieve coverage over all feasible nondeterministic matches by tracking causality (or, the happens-before order) instead of relying upon real-time delays. In addition, we show how the traditional notion of happens-before is not sufficient to capture causality within an MPI execution. Our key contribution is to precisely formulate the matches-before relation underlying MPI execution and exploit it to discover alternative nondeterministic matches. We show that our methods can work at large scale based on efficient runtime mechanisms that enforce feasible matches while avoiding impossible matches.

In addition to computing causal orderings correctly in the presence of nondeterminism for the wide range of MPI constructs, a practical MPI debugger must scale with the application, i.e., potentially to several thousands of MPI processes. This requirement is essential since many test inputs require a large process count to run. Downscaling is often impossible or can mask the bug.

In this paper, we detail how the causal ordering between MPI nondeterministic receive calls and MPI send calls is determined efficiently using a new protocol, the Lazy Lamport Clocks Protocol (LLCP). We do not detail how to handle other MPI call types (e.g., MPI nondeterministic probe commands), but the details are similar. Once matches-before is
correctly computed, a dynamic verifier can determine all feasible matches for a nondeterministic receive. The remainder of this paper focuses on how to determine matches-before (or causal ordering) in production-scale MPI programs.

Tracking causality for MPI is not a new idea [10], [15]. These algorithms use logical clocks to detect causality (or the lack thereof) between send and receive events. However, as detailed later, most algorithms can only handle a small subset of MPI and also do not scale well. We are not aware of any prior work that can handle the example in Figure 1, which is a fairly simple MPI communication pattern. Further, all surveyed work uses vector clocks, despite their high overhead because the much cheaper Lamport clocks are well known to not provide sufficient precision to determine causality.

We show that while our LLCP algorithm is based on Lamport clocks, it still exhibits an extremely low omission probability (none in all real benchmarks that we have studied) because it exploits two additional crucial pieces of information that are not exploited in traditional Lamport clock-based approaches: (i) the matches-before formal model of MPI, and (ii) observed matches during an initial MPI run to infer causalities within alternate runs. Our LLCP algorithm can also be implemented using vector clocks, should one become aware of a concern on real examples; in that case, it still offers significant advantages over standard vector-clock based algorithms that, by tracking all events, incur a high overhead. In contrast, our LLCP algorithms track only relevant events which, in our case are nondeterministic receives. This further improves the scalability of our approach. While we express our ideas in terms of MPI, they apply to any computational model that satisfies the conditions that we describe in Section III-A.

To summarize, we contribute an algorithm based on Lamport clocks that uses lazy updating rules to model the behavior of MPI nonblocking calls properly and to enable detection of matches that prior work misses. We also provide a vector clocks extension of the LLCP algorithm (lazy vector clocks) and evaluate the performance of the two protocols. Finally, we formally characterize both algorithms.

II. BACKGROUND

The usage of logical clocks to track causality in distributed systems has been extensively studied and enhanced. Most of approaches are either based on Lamport clocks [11] or vector clocks [5], [12].

**Lamport clocks:** The Lamport clock mechanism is an inexpensive yet effective mechanism to capture the total order between events in distributed systems. We briefly summarize the algorithm. Each process $P_i$ maintains a counter $LC_i$ initialized to 0. When event $e$ occurs, $P_i$ increments $LC_i$ and assigns the new value, $e.LC$, to $e$. If $e$ is a send event, $P_i$ attaches $e.LC$ to the message. On the other hand, if $e$ is a receive event, $P_i$ sets $LC_i$ to a value greater than or equal to its present value and greater than that which it receives.

Let $e^n_i$ be the $n^{th}$ event that occurs in process $P_i$, $send(P_i, msg)$ be the event corresponding to sending message $msg$ to $P_i$, and $recv(P_i, msg)$ be the event corresponding to the reception of $msg$ from $P_i$. We now define the happens-before ($\langle \text{hb}, \rangle$) relationship over all events:

$$e^n_i \langle \text{hb}, \rangle e^n_j \text{ when :}$$

- $i = j \land n > m$ (local event order)
- $e^n_i = send(P_j, msg) \land e^n_j = recv(P_i, msg)$ (send/receive order)
- $\exists j : e^n_i \langle \text{hb}, \rangle e^n_k \land e^n_k \langle \text{hb}, \rangle e^n_j$ (transitive order)

Given the Lamport clock algorithm described above, we can infer the clock condition: for any two events $e_1$ and $e_2$, if $e_1 \langle \text{hb}, \rangle e_2$ then $e_1.LC < e_2.LC$. Henceforth, we use $\langle \text{hb} \rangle$ instead of $\langle \text{hb}, \rangle$ when the context is clear.

**Vector clocks:** The timestamping algorithm proposed by Lamport has one major limitation: it does not capture the partial order between events. To address this problem, Fidge [5] and Mattern [12] independently propose that the single counter on each process $P_i$ should be replaced by a vector of clocks with size equal to the number of processes in the execution. The vector clock algorithm ensures the bi-implication of the clock condition, that is, for any two events $e_1$ and $e_2$, $e_1 \langle \text{hb,vc} \rangle e_2$ if and only if $e_1.VC < e_2.VC$. Here, $e_1.VC < e_2.VC$ means that for all $i$, $e_1.VC[i] \leq e_2.VC[i]$, and furthermore, there exists a $j$ such that $e_1.VC[j] < e_2.VC[j]$. The clock update rules for vector clocks are as follows. Initially, for all $i$, $VC_i[j]$ is set to 0. Upon event $e$ in $P_i$, $P_i$ increments $VC_i[j]$ and associates it with $e$ ($e.VC$). When $P_i$ sends a message $m$ to $P_j$, it attaches $VC_i$ to $m$. $VC_i[j]$ and associates it with $e$ ($e.VC$). When $P_i$ receives $m$, it updates its clock as follows: for all $k$, $VC_i[k] = \max(VC_i[k], VC[m][k])$. If two events’ vector clocks are incomparable, the events are considered concurrent.

However, this increased accuracy comes at a price: the algorithm must store and send the full vector clocks, which incurs significant overhead at large scales. While some optimizations, like vector clock compression, exist, the worst case behavior of vector algorithms is still unsalable.

**Piggybacking:** All causality tracking protocols discussed above rely on the ability of the system to transmit the clocks together with the intended user data on all outgoing messages. We refer to the transmission of the clocks as piggybacking and the extra data (i.e., the clocks) as piggyback data. We can transmit piggyback data in many different ways such as packing the piggyback data together with the message data, or sending the piggyback data as a separate message [16]. The details of those approaches are beyond the scope of this paper and are thus omitted.

III. MATCHES-BEFORE RELATIONSHIP IN MPI

A. Our Computational Model

A message passing program consists of sequential processes $P_1, P_2, ..., P_n$ communicating by exchanging messages through some communication channels. The channels are assumed to be reliable and support the following operations:

- $send(\text{dest}, T)$ - send a message with tag $T$ to process $\text{dest}$. This operation has similar asynchronous semantics to MPI_Send: it can complete before a matching receive has been posted.
recv(src, T) - receive a message with tag T from process src. When src is MPI_ANY_SOURCE (denoted as *), any incoming message sent with tag T can be received (a wildcard receive).

isend(dest, T, h) - the nonblocking version of send with request handle h.

irecv(src, T, h) - the nonblocking version of recv with request handle h.

wait(h) - wait for the completion of a nonblocking communication request, h. According to MPI semantics, relevant piggyback information for a nonblocking receive is not available until the wait call. Similarly, for a nondeterministic nonblocking receiving, the source field (identity of the sender) is only available at the wait.

barrier - all processes must invoke their barrier calls before any one process can proceed beyond the barrier.

For illustrative purposes, we ignore the buffer associated with the send and recv events, as the presence of buffering does not affect our algorithm. Further, we assume that all the events occur in the same communicator and that MPI_ANY_TAG is not used. We also do not consider other collective operations other than MPI_BARRIER. Our implementation, however, does cover the full range of possibilities.

B. Issues with Traditional Logical Clock Algorithms

We briefly discuss how one could apply the traditional vector clock algorithm to the example in Figure 2 to conclude that the first wildcard receive in P0 can match either send from P1 and P2 (and also why Lamport clocks fail to do do).

We assume that the first receive from P0 matches the send from P1 and that the second receive from P0 matches the send from P2. We must determine whether the vector clock algorithm could determine that the first receive from P0 could have received the message that P2 sent. Using the clock updating rules from the vector clock algorithm, the vector timestamp of P0’s first receive is [1, 1, 0] while that of the send from P2 is [0, 2, 2]. Clearly, the send and the receive are concurrent and thus, the send could match the receive.

In contrast, if we apply the original Lamport clock algorithm to this example, P0’s first receive event would have a clock of 1 while the send from P2 would have a clock of 3. Thus, the algorithm cannot determine that the two events have no causal relationship and cannot safely identify the send from P2 as a potential match to the first receive from P0. One can observe that the communication between P1 and P2 in this example has no effect on P0, yet the matching causes a clock increase that disables P0’s ability to determine the causality between the first wildcard receive and P2’s send.

Now consider the example shown in Figure 1. Assuming the irecv by P1, denoted as r, matches the isend from P0, we apply the vector clock algorithm to determine if the isend from P2, denoted as s, can match r. By using the vector clock updating rules and considering barrier as a synchronization event in which all processes synchronize their clocks to the global maximum, the clocks for r and s are [1, 0, 0], and [1, 0, 1], respectively. Thus, r \( \triangleright_{\text{hnc}} \) s and the algorithm fails to recognize s as a potential match to r.

Clearly, the notion of happening and the corresponding happens-before relationship are insufficient in capturing all behaviors of MPI programs. We need a new model that completely captures the ordering of all events within an MPI program execution.

C. The Matches-before Relationship

We first consider all possible states of an MPI operation op after a process invokes op:

- issued - op attains this state immediately after the process invokes it. All MPI calls are issued in program order.
- matched - We define this state in Definition 3.1.
- returned - op reaches this state when the process finishes executing the code of op.
- completed - op reaches this state when op no longer has any visible effects on the local program state. All blocking calls reach this state immediately after they return while nonblocking calls reach this state after their corresponding waits return.

Of these states, only the issued and matched states have significant roles in our algorithms; nonetheless, we included all possible states for completeness. The matched state is central to our protocols. We describe it in detail below.

Definition 3.1: An event e in an MPI execution attains the matched state if it satisfies one of these conditions:

- e is the send event of message m and the destination process has started to receive m through some event e’. e is said to have matched with e’. The receiving process is considered to have started to receive m when we can (semantically) determine from which send event it will receive data. The timing of the completion of the receiving process is up to the MPI runtime and is not relevant to this discussion. In this case, we consider e and e’ to be in a send-receive match-set.
- e is a receive event that marks the start of reception.
- e is a wait(h) call and the pending receive request associated with h has been matched. For an isend, the wait can attain the matched state upon completion while the isend still has not matched (i.e., it is buffered by the MPI runtime). A matched wait is the only element in its match-set (a wait-match-set).
- $e$ is a barrier and all processes have reached their associated barrier. All participating barriers belong to the same match-set (a barrier match-set). We consider $e$ to have matched $e$ if they are in the same barrier match-set.

While determination of the matching point of `recv` and barrier calls is straightforward, the situation is more complex with nonblocking calls. The assumption that all nonblocking calls attain the matched state exactly at their corresponding `wait` call returns, which could be much earlier than the completion of its corresponding `wait`. This interval arises from the nonovertaking rule of the MPI standard, which we describe later in this section.

Let $E$ be the set of events produced in an execution $P = \langle E, \mb \rangle$, where each $e \in E$ is a match event and $\mb$ is the matches-before relation over $E$ defined as follows: consider two distinct events, $e_1, e_2 \in E$, $e_1 \mb e_2$ if and only if one of the following conditions holds:

- **C1.** $e_1$ is a blocking call (send, `recv`, wait, barrier) and $e_2$ is an event in the same process that is issued after $e_1$.
- **C2.** $e_2$ is the corresponding `wait` of nonblocking call $e_1$.
- **C3.** $e_1$ and $e_2$ are send events from the same process $i$ with the same tag, targeting the same process $j$ and $e_1$ is issued before $e_2$. This is the nonovertaking rule of MPI for sends. The sends can be blocking or nonblocking.
- **C4.** $e_1$ and $e_2$ are receive events from the same process $i$ with the same tag, either $e_1$ is a wildcard receive or both are receiving from the same process $j$, and $e_1$ is issued before $e_2$. This is the nonovertaking rule of MPI for receives. The receives can be blocking or nonblocking.
- **C5.** $e_1$ and $e_2$ are from two different processes and $\exists e_3, e_4 \in E$ in the same match-set and $e_3$ is not a receive event (i.e., $e_3$ is a send, `isend`, or `barrier`) such that $e_1 \mb e_3, e_2 \mb e_2$. Figure 4 illustrates this transitivity condition. The two shaded areas in the figure show a send-receive match-set and a barrier match-set while the dashed arrows show the matches-before relationship between events in the same processes. Condition C5 implies that $e_1 \mb e_2$ and $e_2 \mb e_3$.
- **C6.** $\exists e_3 \in E$ such that $e_1 \mb e_3$ and $e_3 \mb e_2$ (transitive order). In Figure 4, condition C6 combines with C5 to imply that $e_1 \mb e_3$.

**Corollary 3.1:** If $e_1$ and $e_2$ are two events in the same match-set, neither $e_1 \mb e_2$ nor $e_2 \mb e_1$ holds.

**Corollary 3.2:** If $e_1$ and $e_2$ are two events in the same process and $e_1$ is issued before $e_2$, then $e_2 \mb e_1$ is false. This is directly based on the definition of matches-before.

In addition to the $\mb$ relationship for two events, we also define the $\mb$ relationship between $X$ and $Y$ where either $X, Y$, or both are match-sets: $X \mb Y$ if and only if one of the following conditions holds:

- **C7.** $X$ is an event, $Y$ is a send-receive match-set, $e_2$ is the send event in $Y$, and $e_1 \mb e_2$.
- **C8.** $X$ is an event, $Y$ is a barrier match-set or a wait match-set, and for each event $e_2 \in Y$, $e_1 \mb e_2$.
- **C9.** $X$ is a send-receive match-set, $e_1$ is the receive event in $X, Y$ is an event $e_2$, and $e_1 \mb e_2$.
- **C10.** $X$ is a barrier match-set or a wait match-set and $Y$ is an event $e_2$, and $\exists e_1 \in X$ such that $e_1 \mb e_2$.
- **C11.** $X$ and $Y$ are match-sets and $\exists e_1 \in X$ such that $e_1 \mb Y$.

**Concurrent events:** We consider $e_1$ and $e_2$ concurrent if they are not ordered by $\mb$. Let $e_1 \mb e_2$ denote that $e_1$ is not required to match before $e_2$; then $e_1$ and $e_2$ are concurrent if and only if $e_1 \mb e_2 \land e_2 \mb e_1$.

**IV. Lazy Update Algorithms**

Since we have adopted the matches-before relationship from the traditional Lamport happens-before, we need new clock updating algorithms that capture the causality information between events in an MPI execution based on $\mb$. To this end, we propose two algorithms: the Lazy Lamport Clocks Protocol (LLCP) and the Lazy Vector Clocks Protocol (LVCP). These are our design goals:

- **scalable** - the protocol should scale to large process counts;
- **sound** - the protocol should not indicate that impossible matches are potential matches;
- **complete** - the protocol should find all potential matches.

We require a sound protocol while the other goals are desirable. An unsound protocol can cause a deadlock in an otherwise deadlock-free MPI program. Both LLCP and LVCP are sound. Many MPI applications today require large scale runs in order to test important input sets due to memory size and other limits. Also, many bugs, including nondeterminism
related bugs, are only manifest at large scales. Thus, a protocol
must scale to support finding many important errors. LLCP
is scalable compared to LVCP as our experimental results
demonstrate. However LVCP is complete while LLCP is not.
The design of a complete and scalable protocol represents
a significant challenge. We require a scalable protocol that
maintains completeness for common usage. In our testing with
real MPI programs, LLCP is complete, that is we did not
discover any extra matches when we ran the same program
under LVCP. When completeness is absolutely required, one
should use LVCP at the cost of scalability.

A. The Lazy Lamport Clocks Protocol

Relevant events: Clearly, not all events in an MPI execution
are relevant to our purposes of tracking causality. However,
one cannot naively track only relevant events and ignore
irrelevant events because these events can indirectly introduce
causality between relevant events. In our protocol, we con-
sider wildcard receives (whether blocking or nonblocking) as
relevant events. While we do not consider sends as relevant,
our algorithm still allows us to track causality between sends
and relevant events.

Algorithm overview: LLCP maintains the matches-before
relationship between events by maintaining a clock in each
process and associates each event with a clock value so we can
order these events according to when they attain the matched
state. Since the matches-before relationship describes the
ordering for events within and across processes, the algorithm
needs must provide such coverage. More specifically, given a
wildcard receive r from process P_i and a send s targeting
P_i that did not match r, the protocol should allow us to
determine if r and s have any matches-before relationship. If,
for example, the successful completion of r triggers the issue
of s, then s could never match r. The intuitive way to achieve
this soundness is for the protocol to maintain the clock such
that if r triggers the issue of some event e then the clock of r
must be smaller than the clock of e. Basically this condition
requires all outgoing messages after r from P_i to carry some
clock value (as piggyback data) higher than r.

The challenge of the protocol lies in the handling of
nonblocking wildcard receives. As explained earlier in the
example in Figure 1, a nonblocking wildcard receive from a
process and associates each event with a clock value so we can
order these events according to when they attain the matched
state. Since the matches-before relationship describes the
ordering for events within and across processes, the algorithm
needs must provide such coverage. More specifically, given a
wildcard receive r from process P_i and a send s targeting
P_i that did not match r, the protocol should allow us to
determine if r and s have any matches-before relationship. If,
for example, the successful completion of r triggers the issue
of s, then s could never match r. The intuitive way to achieve
this soundness is for the protocol to maintain the clock such
that if r triggers the issue of some event e then the clock of r
must be smaller than the clock of e. Basically this condition
requires all outgoing messages after r from P_i to carry some
clock value (as piggyback data) higher than r.

The challenge of the protocol lies in the handling of
nonblocking wildcard receives. As explained earlier in the
example in Figure 1, a nonblocking wildcard receive from a
process P_i could potentially be pending (not yet have reached
the matched state) until its corresponding wait is posted.
However, we have also shown in Figure 3 that the receive
could also attain the matched state due to the nonovertaking
semantics (which could be earlier than the posting of the wait).
The protocol must precisely determine the status of the receive
to avoid sending the wrong piggyback data, which could lead
to incorrect matching decisions. To achieve this precision, we
use the following set of clock updating rules:

- R1. Each process P_i keeps a clock LC_i, initialized to 0.
- R2. When a nonblocking wildcard receive event e occurs,
  assign LC_i to e.LC and add e to the set of pending
  receives: Pending ← Pending ∪ {e}.
- R3. When P_i sends a message m to P_j, it attaches LC_i
  (as piggyback data) to m (denoted m.LC)
- R4. When P_i completes a receive event r (either forced
  by a blocking receive or at the wait of a nonblocking
  receive as in Figure 3), it first constructs the ordered set
  CompleteNow as follows: CompleteNow = \{e | e ∈ Pending ∩ e mb \ r\}. The set CompleteNow is ordered
  by the event’s clock, where CompleteNow[i] denotes the
  i-th item of the set and |CompleteNow| denotes the total
  items in the set. Intuitively, these pending nonblocking
  receives have matched before r due to the nonovertaking
  rule. Since they have all matched, we must update their
clocks. The ordering of the events in CompleteNow is
  important since all receives in CompleteNow are also
  \( mb \) ordered by the nonovertaking semantics. We can
  update the clocks using the following loop:
  for i = 1 TO |CompleteNow|
  CompleteNow[i].LC = LC_i
  \( LC_i ← LC_i + 1 \)
  end for

  Pending ← Pending \ \\ CompleteNow

After this, the process associates the current clock with
r: r.LC ← LC_i and advance its clock to reflect the
completion of a wildcard receive: \( LC_i ← LC_i + 1 \).
The clock assignment and advancement does not happen to
those nonblocking receives that have their clocks increased earlier due to the for loop above. We can check
this condition by detecting if the current nonblocking
receive is still in the Pending set. Finally, the process
compares its current clock with the piggybacked data
from the received message and updates \( LC_i \) to m.LC
if the current clock is less than \( m.LC \).
- R5. At barrier events, all clocks are synchronized to
  the global maximum of the individual clocks.

Lazy Update: Rules R2 and R4 form the lazy basis of the
protocol in the sense that a nonblocking wildcard receive r has
a temporary clock value when it initially occurs and obtains its
final clock value when it finishes (either by its corresponding
wait or by another receive r’ for which r mb r’).

Lemma 4.1: If e_1 mb e_2 then e_1.LC ≤ e_2.LC
Proof: We first consider the case when e_1 and e_2 are from
the same process. Based on our definition of matches-before,
both will always occur after event e_1. Since our algorithm
never decreases the clock, it follows that e_1.LC ≤ e_2.LC

Now assume e_1 and e_2 are events from different processes.
From the definition of matches-before, events e_3 and e_4 exist
such that e_1 mb e_3, e_4 mb e_2, e_3 and e_4 are in a match-set,
and e_3 is an isend, a send, or a barrier. We recursively
apply this process to (e_1, e_3) and (e_4, e_2) to construct the set
S = s_1, s_2, ..., s_n in which s_1 = e_1, s_n = e_2, and other
elements are events or match-sets that satisfy s_i mb s_{i+1}.
Also, S must satisfy the following rule: for any pair of adjacent
elements (s_i, s_{i+1}), no event e exists such that s_i mb e and
e mb s_{i+1}. The construction of S is possible based on our

Definition 4.1: A message \( m \) originating from process \( m.src \) with timestamp \( m.LC \) is considered *late* with respect to a wildcard receive event \( r \) (which earlier did not match with \( m.src \)) if \( m.LC \leq r.LC \). If message \( m \) is late with respect to \( r \), it is denoted as \( \text{late}(m, r) \).

We can now present our match detection rule.

**Theorem 4.3:** An incoming send \( s \) carrying message \( m \) with tag \( \tau \) that is received by event \( r' \) in process \( P_i \) is a potential match to a wildcard receive event \( r \) with tag \( \tau \) issued before \( r' \) if \( (m.LC < r'.LC \land \text{late}(m, r)) \)

**Proof:** In order to prove that \( s \) is a potential match of \( r \), we prove that \( s \) and \( r \) are concurrent, that is, \( r \rightarrow s \) and \( s \rightarrow r \). First, note that \( r \rightarrow s \), which means \( r \) cannot be a pending event (due to rule R4). We also have \( r.LC \geq s.LC \) since \( s \) is a late message. Using the contrapositive of Lemma 4.2, we infer that \( r \not\rightarrow s \). Also, \( s \rightarrow r \) because if \( s \rightarrow r \) then \( s \rightarrow r' \) due to the transitive order rule of matches-before. This conclusion violates Corollary 3.1: \( \rightarrow \text{mb} \) does not order two events in the same match-set.

We now revisit the example in Figure 1. Using the LLCP clock update rules, \( P_1 \) has a clock of 0 when issues its irecv, which it adds to Pending. The barrier calls synchronize all clocks to the global maximum, which is 0. At the recv call, \( P_1 \) applies rule R4 and constructs the CompleteNow set consisting of irecv. Upon the completion of this step, the irecv has a clock of 0 and the recv has a clock of 1. Assuming that the isend from \( P_1 \) matches the irecv, which means the recv call matches the isend from \( P_2 \). The message from \( P_2 \) carries a piggyback data of 0 so it is flagged as a late message with respect to the irecv and is detected as a potential match (per Theorem 4.3). Figure 6 shows the clock values of this execution.

LLCP can miss potential matches. Consider the example in Figure 7 for which manual inspection easily finds that the send from \( P_0 \) is a potential match of the wildcard receive from \( P_2 \) (assuming the \( P_2 \)'s recv matches with \( P_1 \)'s send). However, LLCP would not detect this potential match since at the time of receiving \( P_0 \)'s send, the clock of \( P_0 \)'s send is 1, which is the same as the clock of \( P_2 \)'s recv(0), so it does not satisfy the condition of Theorem 4.3.

This issue again reflects the disadvantage of Lamport clocks given multiple concurrent sources of nondeterminism. In general, omissions may happen with multiple sources of nondeterminism and communication between processes with clocks that are out of synchronization. Fortunately, this situation rarely happens in practice because most MPI programs have well-established communication patterns that do not have cross communications between groups of processes that generate
relevant events before clock synchronization occurs. We will present our extension of LLCP to vector clocks to address MPI programs with subtle communication patterns for which LLCP may not determine all potential matches.

**Handling synchronous sends:** The MPI standard supports synchronous point-to-point communication with MPI_Ssend and MPI_Issend. In order to handle these synchronous calls, we extend the definitions of $\text{mb}$ and LLCP. Specifically, we modify condition C9 to condition $C9'$:

- $C9'$: $e_1$ is a synchronous send in match-set $X$, $e_2$ is the corresponding receive in $X$, and $e_3$ is some event. $X \xrightarrow{\text{mb}} e_3$ if and only if $e_1 \xrightarrow{\text{mb}} e_3 \land e_2 \xrightarrow{\text{mb}} e_3$.

Typically when a receive operation and a synchronous send operation match, the MPI implementation sends an acknowledgment from the receiver to the sender. LLCP handles synchronous sends by modifying the MPI runtime to add piggyback data to this acknowledgment. Since synchronous calls are rarely used, we omit the implementation details.

**B. The Lazy Vector Clocks Protocol**

We extend LLCP to LVCP, which uses vector clocks. LVCP has similar clock updating rules that we modify to apply to vector clocks (e.g., instead of incrementing a single clock, $P_i$ now increments $V C_i[i]$). We now prove the updated lemmas and theorems that are based on LVCP.

**Lemma 4.4:** If $r$ is a blocking receive or a nonblocking receive not pending in $P_i$, then $r \xrightarrow{\text{mb}} e \iff r.V C[i] < e.V C[i]$.

**Proof:** We omit the proof for $r \xrightarrow{\text{mb}} e \implies r.V C[i] < e.V C[i]$, which is similar to the LLCP case. We now prove the converse. Observe that in LVCP as in all vector clock protocols, the only process that increments $V C[i]$ is $P_i$. Thus, $e$ is an event that occurs in $P_i$ after $r$ completes (which is the point at which $V C[i]$ becomes greater than $r.V C[i]$) or it is an event in another process that receives the piggybacked value of $V C[i]$ from $P_i$ (directly or indirectly). If $e$ is an event that occurs in $P_i$ after $r$ completes and $r$ is a blocking receive then $r \xrightarrow{\text{mb}} e$ due to the definition of $\text{mb}$, Alternatively, if $r$ is nonblocking receive, $P_i$ only increases its clock (i.e., $V C[i]$) under one of the following conditions:

- The corresponding wait for $r$ is posted and $r$ is still pending before the wait call.
- A blocking receive $r'$ that satisfies $r \xrightarrow{\text{mb}} r'$ is posted and $r$ is still pending before $r'$.

- A wait for a nonblocking receive $r'$ that satisfies $r \xrightarrow{\text{mb}} r'$ is posted and $r$ is still pending before $r'$.

In all of these scenarios, we need a blocking operation $b$ such that $r \xrightarrow{\text{mb}} b$ in order to increase the clock of $P_i$. If $e.V C[i] \geq r.V C[i]$, $e$ must occur after $b$. Thus, $b \xrightarrow{\text{mb}} e$ and by the transitive order rule, $r \xrightarrow{\text{mb}} e$.

Using the updated definition of late message (Definition 4.1) where $m.LC \leq r.LC$ is replaced by $m.V C[i] \leq r.V C[i]$, we now prove the LVCP matching theorem, which states that all potential matches are recognized.

**Theorem 4.5:** An incoming send $s$ carrying message $m$ with tag $\tau$ that is received by event $r'$ in process $P_i$ is a potential match to a wildcard receive event $r$ with tag $\tau$ issued before $r'$ if and only if $(m.V C[i] < r'.V C[i] \land \text{late}(m, r))$.

**Proof:** We omit the proof for the if part, which is similar to the proof of Theorem 4.3. We now prove the converse. Observe that due to the nonovertaking rule, $r \xrightarrow{\text{mb}} r'$, which gives us $r.V C[i] < r'.V C[i]$ as asserted. First we notice that $m.V C[i] < r'.V C[i]$ then $m.V C[i] < r'.V C[i] \land \text{late}(m, r)$. We apply Lemma 4.4 to $r \xrightarrow{\text{mb}} s$, to obtain $m.V C[i] < r'.V C[i]$, which means $m.V C[i] < r'.V C[i] \land \text{late}(m, r)$ (note that $m.V C[i]$ is the same as $s.V C[i]$ since $m.V C[i]$ is the piggyback value attached due to $s$).

**V. EXPERIMENTAL RESULTS**

We implement LLCP and LVCP as separate $P^{\text{N}}\text{MPI}$ [17] modules and integrate them in DAMPI, which is our distributed analyzer of MPI programs. DAMPI analyzes the program for freedom of deadlocks, resource leaks, and assertion errors. DAMPI provides the unique ability to guarantee coverage over the space of nondeterminism through replay. When either protocol (LLCP or LVCP) detects alternative matches to nondeterministic receives, DAMPI records the matches so that a scheduler can analyze the matches offline to generate a decision database that the processes parse during replay. The decision database contains the information that the processes need to replay themselves correctly up until the branching point (i.e., the wildcard receive that should match a different source during the particular replay) and rewrite the $\text{MPI\_ANY\_SOURCE}$ field into the specified value. The replay mechanism of the scheduler is beyond the scope of this paper. More details on DAMPI in general and its scheduler in particular are available in a related dissertation [22].

**Experimental Setup:** We run our benchmarks on the Atlas cluster at Lawrence Livermore National Laboratory. This Linux-based cluster has 1152 compute nodes, each with 8 cores and 16GB of memory. All experiments were compiled and run under MVAPICH2-1.5 [1].

We first report on the latency and bandwidth impact of the two protocols. For latency testing, we use the OSU multipair latency benchmark [1] and report the latency testing result for 4-byte messages as the number of processes in the job increases. These small messages represent the class of messages for which latency impact is most significant. On
the other hand, for bandwidth testing, we use a simple ping-pong test between two processes in the system, while others sit idle. While typical bandwidth tests report the bandwidth as the message size grows, such tests do not consider the number of processes in the system and thus do not capture the scalability of the protocols. Thus, we report the message sizes at which the system achieves half of the peak bandwidth for the bandwidth testing (the R/2 value).

Figures 8 and 9 summarize our latency and bandwidth testing results. Higher bars indicate worse performance since they correspond to longer latencies and to larger message sizes to achieve the same bandwidth. The results show that both protocols have manageable latency at low process counts but the impact of vector clocks becomes more significant as the system scales up while LLCP maintains stable latency penalties across the entire range. At 1024 processes, the latency of messages under LLCP increases by only 10% compared to the original, uninstrumented messaging while it increases 240% under LVCP. Both protocols achieve essentially the same peak bandwidth as uninstrumented messaging but reduce the bandwidth achieved at intermediate message sizes. Importantly, this impact is more pronounced with LVCP and increases with increasing process count, while the impact of LLCP is independent of the process count.

Latency and bandwidth do not always translate to overhead since programs typically do not exchange messages 100% of the time. Therefore, we evaluate the performance of three scientific MPI applications: ParMETIS, a parallel hypergraph partitioning library [9]; AMG2006, an algebraic multi-grid solver for unstructured mesh available from the ASC Sequoia benchmark [2]; and SMG2000, a semicoarsening multi-grid solver from the ASCII Purple benchmark [3]. These applications are designed to run at large scales so we vary the
process count from 64 to 1024 (8 processes per node). We run ParMETIS using the supplied testing parameters. We run both AMG2006 and SMG2000 with a 6x6x6 grid as input. We report the average time for five runs. ParMETIS and SMG2000 do not have any nondeterministic receives and thus the results reflect the overhead of checking for common MPI errors (e.g., resource leaks). AMG2006 has wildcard receives and wildcard probes (the protocols can handle MPI_Probe and MPI_Iprobe; we omit the details). We report the performance for the first run, which reflects the overhead of tracking wildcard events and checking for errors. LLCP and LLVP find the same potential matches even though AMG2006 issues wildcard receives from multiple processes, which confirms that practical applications do not lead to omissions under LLCP.

AMG2006 and SMG2000 display similar trends in which LVCP remains competitive with LLCP until we use 1024 processes, when LLCP provides a significant advantage. We expect the difference to increase with more processes due to the larger vector clocks. For ParMETIS, which issues almost 550 million messages (compared to AMG at 200 million and SMG2000 at 40 million) under our experiment with 1024 processes, over 98% of these messages are between 4-256 bytes and thus LVCP suffers a high latency penalty to transmit the vector clocks. We also notice several interesting cases in which LLCP has negative overhead (i.e., it runs faster with extra processing), which probably arises from the small extra payload altering the communication pattern to result in better optimization from the MPI runtime.

One issue worth noting but not conveyed from the figures is the memory overhead associated with LVCP. For each relevant event, LVCP must keep a vector clocks in order to track the causality of the event, which results in large memory usage for bookkeeping, especially when the programs have many wildcards and run at large scale. For example, AMG2006 generated about 1800 relevant events per process with 1024 processes, which results in about 7 Gigabytes of extra memory (collectively) to keep track of vector clocks. As we run the programs at much larger scales, this memory overhead will be prohibitively expensive.

Compressed vector clocks [8], [20] can reduce the memory usage and the bandwidth overhead of vector clocks. Some proposed schemes do not directly apply to MPI due to their special requirements (e.g., first-in-first-out channels [20]). Nonetheless, the efficiency of these schemes are highly dependent on the communication pattern and all require more local memory storage for bookkeeping than traditional vector clocks. As supercomputers employ more and more cores, the amount of physical memory per core is decreasing so this additional overhead could prevent the tools from running. Further, all compressed vector clocks schemes send a variable amount of piggyback information, including at collective calls. Since most MPI implementations do not yet support native piggybacking, implementing a variable piggybacking layer forces the developer to use explicit buffer packing, which greatly increases performance overhead (DAMPI currently uses datatype piggybacking, which has lower overhead compared to explicit buffer packing). Also, one would need to break collective calls into pair-wise calls to support variable message sizes and thus forfeit any MPI collective optimization or to send the entire vector clock for collectives.

VI. RELATED WORK

Schedule control techniques are routinely used to analyze thread-based concurrent programs. These techniques range from sophisticated dynamic partial order methods [6], [25] to schedule bounding techniques [4], [14] to randomized search that relies on bug depth [13]. Unfortunately, none of these methods can apply directly to MPI programs [21], [24] because message passing (and MPI in particular) programs have different methods of interactions compared to shared memory programs as indicated in our discussion of MPI's matches-before model.

While some works have attempted to use logical clocks to track nondeterminism such as MPIRace-Check [15] and NOPE [10], most rely on the traditional vector clock algorithm and, thus, do not capture the full semantics of MPI. We have run MPIRace-Check on the example in Figure 1 and it failed to report the match (the source code for NOPE was not available at the time of our experiments).

ISP [21], [24] is a dynamic scheduling control algorithm that provides formal coverage guarantees. However, we showed that ISP’s centralized schedule generation and control algorithms do not scale well [23].

DAMPI [23] is our dynamic verifier for MPI programs that uses an eager update mechanism to maintain causality between events. We earlier showed that eager clock updates can lead to high omission rates (in particular, the 'crooked barrier' example was shown to be outside its scope). This paper introduces the notion of matches-before which provides the basis for lazy protocols. We formally show that lazy protocols can correctly capture the full semantics of MPI with modest performance overhead.
MPI-SPIN is a model checker for MPI that can help to verify the concurrency and synchronization skeleton of MPI program models formally [18], [19]. However, this approach requires the user to build the model for the MPI program manually, which is not practical for large scale MPI programs.

VII. CONCLUSIONS

The MPI standard offers a rich set of features such as non-blocking primitives and nondeterministic constructs that help developers write high performance applications. These features, however, complicate the task of large-scale debugging, especially over the space of nondeterminism, which requires causality tracking. Traditional causality tracking algorithms, such as Lamport clocks and vector clocks, are usually not sufficient to handle its complex semantics. In this paper we propose the lazy Lamport clocks protocol and its vector clock variant that can correctly track nondeterminism in MPI programs. The lazy update scheme correctly models nonblocking MPI calls and thus can track nondeterminism more precisely. A scheduler can then use the traced information to perform replays to explore different executions of the program. We have integrated our proposed approach into DAMPI, thus enabling the ability to track and to replay nondeterministic operations to detect common MPI usage errors. We have tested DAMPI with large and practical MPI programs on up to a thousand processes. Our experiments show that the lazy Lamport clocks protocol is scalable and provides complete coverage in practice.

REFERENCES