A Predictive Analysis for Detecting Deadlock in MPI Programs

Yu Huang
Southwestern University of Finance and Economics
yuhuang@swufe.edu.cn

Benjamin Ogles
Brigham Young University
benjaminogles@gmail.com

Eric Mercer
Brigham Young University
degm@cs.byu.edu

ABSTRACT
A common problem in MPI programs is deadlock: when two or more processes are blocked indefinitely due to a circular communication dependency. Automatically detecting deadlock is difficult due to its schedule-dependent nature. This paper presents a predictive analysis for single-path MPI programs that observes a single program execution and then determines whether any other feasible schedule of the program can lead to a deadlock. The analysis works by identifying problematic communication patterns in a dependency graph to form a set of deadlock candidates. The deadlock candidates are filtered by an abstract machine and ultimately tested for reachability by an SMT solver with an efficient encoding for deadlock. This approach quickly yields a set of high probability deadlock candidates useful for reasoning about complex codes and yields higher performance overall in many cases compared to other state-of-the-art analyses. The analysis is sound and complete for single-path MPI programs on a given input.

CCS CONCEPTS
- Software and its engineering → Formal software verification. Distributed programming languages.

KEYWORDS
deadlock, predictive analysis, abstract machine, cycle detection

ACM Reference Format:

1 INTRODUCTION
The message passing interface (MPI) is the de facto standard for communication and synchronization in high performance distributed programs. MPI programs contain a finite set of processes that send and receive messages concurrently. A common error in MPI programs, referred to as deadlock, occurs when one or more processes block indefinitely due to a circular communication dependency. This paper presents an analysis for detecting deadlock in a subclass of MPI programs called single-path programs [10] where the order of actions issued by each process is deterministic for a given input. Single-path programs therefore exhibit a limited, though still significant, set of non-deterministic behaviors due to message race.

Message race occurs when two or more messages are sent concurrently to the same process that has not specified a source for its next receive. An MPI program that terminates successfully in one execution may deadlock in another execution depending on how message race is resolved by the runtime.

The analysis presented in this paper is predictive because it observes a single schedule over message race and finds a different feasible schedule that leads to a deadlock on the same input. The new schedule preserves the order of actions in each process and is therefore guaranteed to be a feasible execution of the single-path program. Finding such a schedule is an NP-Complete problem and can be encoded as a propositional formula [10].

These formulas can grow prohibitively large for many programs because they must encode what it means to deadlock generally. In contrast, the approach presented in this paper only encodes what it means to deadlock at a specific point in the program. This simpler encoding is used on a filtered set of likely deadlock candidates.

The analysis proceeds in three increasingly precise stages:

1. A conservative set of deadlock candidates is extracted from the cycles of a dependency graph defined over the observed execution.
2. Each candidate is tested for reachability in an abstract machine that implements an abstract semantics for the presented MPI programming model.
3. The feasibility of the remaining candidates is determined by encoding each as a separate SMT problem.

Each of these stages presents its own research challenges. First, highly non-deterministic MPI programs yield large dependency graphs with a huge number of cycles. A naïve enumeration of all cycles would not scale to these programs. This paper presents a strict characterization of which cycles may be deadlock candidates and a modification of Johnson’s algorithm [18] that only enumerates such cycles.

Second, to be an effective filtering mechanism, the abstract machine must be able to accept all feasible deadlock cycles and reject most infeasible deadlock cycles based on only one abstract execution. In other words, the machine must be provably precise and the schedule chosen by the abstract machine execution must not lead to spurious results. Proofs for both of these properties are given in the full paper [2].

Finally, the SMT encoding is adapted from an encoding developed by Huang et al [16] that can only describe schedules over message race that form complete executions (e.g., the end of each
We formalize an observed MPI program execution as a concurrent (receive) buffer. The barrier (indicated with a unique identifier) is available (i.e., when data has been copied out of (into) a send (receive) buffer). The messages are held in buffers allocated by the user program and copied from source to destination buffers by an MPI runtime. Data buffers are not explicitly represented in this presentation of the analysis and accompanying proofs, we assume that the last action in each process \( p \in P(A) \) is a barrier action (bip g) for some group \( g \in G(A) \) where \( G(A) \) is the set of barrier groups in A. This assumption does not affect the programs we analyze as we can always extend the observed CTP with these actions. The added barrier action at the end of each process is necessary to model all potential deadlocks in the dependency graph including those arising from wildcard receives.

The disjoint sets of sends, receive, wait and barrier actions in A are denoted respectively by \( S(A), R(A), W(A), B(A) \). For an action \( a \in A \), \( id(a) \) and \( pid(a) \) denote the action and process identifiers of \( a \). For an action \( a \in S(A) \cup R(A) \), \( src(a) \) and \( dst(a) \) denote the source and destination process of \( a \). Of course, \( src(a) = pid(a) \) and \( dst(a) = pid(a) \) for sends and receives respectively.

For an action \( a \in W(A) \), \( req(a) \) is the non-blocking action that \( a \) waits for. For an action \( a \in B(A) \), \( g = grp(a) \) is the group identifier for \( a \) and \( B_p(A) \) is the set of actions in that group.

For a process \( p \in P(A) \), the actions owned by \( p \) are always issued sequentially. This constraint can be captured as a partial order over \( A \) if we assume that action identifiers were assigned in ascending order while observing the original execution.

**Definition 2.1 (Process order).** Process order is a partial order \((A, \leq_{po})\) where

\[
\forall a, a' \in A : a \leq_{po} a' \iff pid(a) = pid(a') \land id(a) \leq id(a')
\]

For Definition 2.1 and similarly for any other partial order defined in this paper, we use \( <_{po} \) to mean the partial order with reflexivity removed and we omit \( A \) from the notation when it is clear from context.

### 3 EXAMPLE

Figure 2 shows a feasible complete execution for a simple CTP. Each column is a separate process where actions are listed in \( \leq_{po} \) order. The vertical spacing represents the total order over the actions in the observed execution. We refer to actions by their unique identifiers.

In this example, the wildcard receive 0 can match with either 1 or 3. In the complete execution 0 matches with 1, leaving 3 to match with 4. If this message race is resolved in the opposite direction, matching 0 with 3, a deadlock occurs.

Our analysis detects the deadlock by identifying the program locations \(((w \ 9 \ 0 \ 6), (w \ 11 \ 14), (w \ 7 \ 2 \ 1))\) as a deadlock candidate. The deadlock candidate is extracted from the cycle shown in Figure 3. Figure 4 shows the witness execution for this deadlock. Here 6 is waiting for its only match: 8. This prevents 10 from being issued and matching with 4, which in turn prevents 12 from being issued and matching with 1.

The graph for Figure 2 contains 22 nodes and 36 edges. Our algorithm detects three deadlock cycles in this graph. In this case, none of the cycles are filtered away by the abstract machine, even though only one predicts a feasible deadlock. The other two cycles predict infeasible deadlocks.
match but process 1 is waiting to finish. This type of deadlock does
predict that the program in Figure 2 can be re-ordered to reach a
state where 7, 9 and 11 are issued and where 1, 4 and 6 cannot
find any match. The satisfying assignment for the variables in the
formula can be used to construct the witness execution, including
the details of which matches were made.

The infeasible cycles report the program locations (b 17 10) and
(w 7 2 1) in two different candidates. In other words, they each
predict that the program in Figure 2 can be re-ordered to reach a
state where action 1 is waiting for process 1 to issue a compatible
match but process 1 is waiting to finish. This type of deadlock does
occur in some programs and is discussed in Section 6. However, in
this case it is clear that if process 1 reaches action 17, as predicted in
the deadlock candidates, either action 9 or 12 must be available to
match with action 1, making these candidates infeasible. It may be
possible to extend the abstract machine with some counting logic
to filter these candidates but our empirical results (Section 9) show
that it already filters a majority of the cycles detected in our set of
benchmarks.

4 SEMANTICS OF CONCURRENT TRACE PROGRAMS

In this section, we extend the semantics presented by Forejt et
al. [10] to accomodate barrier groups and simpler proofs of cor-
rectness for our analysis. The semantics of A is given by a finite
state machine F (A) = ⟨Q, q0, →⟩ where Q ≤ 2^M × 2^M × 2^M is
the set of states, q0 = (Α, 0, 0) is the start state and →⊆ Q × Q is
the transition relation. In a state q = ⟨I, M⟩, A is the set of actions
in the CTP, I ⊆ A is the set of actions that have been issued to the
runtime, and M ⊆ I is the set of actions that have been matched.

Send and receive actions are matched together by the runtime
when their source and destination processes are compatible. Wait
actions do not match other actions but are instead “matched” with
themselves after their associated message request is matched. Fi-

ally, barrier actions match the other actions in their group when
they have all been issued.

Some actions in the same process may be matched in an order
different from how they were issued while others must be matched
in the order they were issued. This constraint is captured by two
more partial orders.

Definition 4.1 (Queue order). Queue order is a partial order (A, ≤q)
where for all actions a, a′ ∈ A a ≤q a′ if and only if a ≤A a′ and one of the following is true:

1. {a, a′} ⊆ S(A) ∧ dst(a) = dst(a′)
2. {a, a′} ⊆ R(A) ∧ src(a) ∈ {src(a′), *}

Figure 4: Witness execution for deadlock in Figure 3.
Definition 4.1 defines a first-in-first-out (FIFO) ordering over messages communicated on the same endpoint. With one exception, this order fully supports the "non-overtaking" property of ordered messages as defined in the MPI standard. The exception occurs when a deterministic receive is followed by a wildcard receive in the same process.

In this case, FIFO ordering over the two receive actions is enforced only if they can both match the same send action. This condition is schedule-dependent as its value changes depending on whether a send action has been issued that can match the first receive action when the second action is issued. As in [10], we leave such receive actions unordered.

Definition 4.2 (Match order). Match order is a partial order \((A, \preceq_{\text{mq}})\) where for all actions \(a, a' \in A\), \(a \preceq_{\text{mq}} a'\) if and only if \(a \preceq a'\) and one of the following is true:

1. \(a \preceq_{\text{so}} a'\)
2. \(a \in W(A) \cup B(A)\)
3. \(a \in S(I(A)) \cup R(A) \land a' \in W(A) \land a = \text{req}(a')\)

Definition 4.2 ensures that (1) queue order is preserved when messages are matched, (2) blocking actions are matched before subsequent actions in the same process and (3) message requests are matched before their associated wait actions. With match order defined, we can define the transition relation \(\rightarrow\) shown in Figure 5. Given a relation \(Q\) over a set \(X\) and an element \(x \in X\), \(Q^{-1}(x) = \{y \in X : y\,Q\,x\}\) is the preimage of \(x\) under \(Q\). The \textit{Issue} transition issues a new action when all of the actions preceding it in the same process have been issued and all of the blocking actions preceding it in the same process have been matched. The \textit{Match-Send-Recv} transitions complete type compatible matches when their match order dependencies have been satisfied.

Note that the definition of the \textit{Match-Wait} transition in \(\rightarrow\) requires that the \textit{Match-Send-Recv} transition has completed for the message request of the wait action. This effectively forces executions to follow a rendevous protocol where waiting for a communication causes the process to block until the full message transfer is completed. This is in contrast to an equally valid protocol where wait actions are matched as soon as the data buffer used in the message request is available.

We choose to use the rendevous protocol because it is equivalent to a zero-buffer runtime. In a buffered runtime, sent messages can be copied into system buffers before matching receive actions are issued in order to free up user buffers more quickly. In a zero-buffer runtime, this buffering never occurs and the user buffer is only available after the message transfer has completed.

Our analysis can also support another important buffer setting for MPI programs: the infinite-buffer setting. In an infinite-buffer runtime, sent messages are buffered without limit. The \textit{Match-Wait} transition is incompatible with this buffer setting. Rather than creating new transition rules however, we completely support the infinite-buffer setting by removing wait actions from the CTP that wait on send actions.

5 DEADLOCK

This section formalizes the problem statement for detecting deadlock in \(A\). The definitions are given in terms of a generic transition relation \(\delta \subseteq Q \times Q\) because we reuse them later with the abstract transition relation.

Let \(\Sigma_0^Q \subseteq Q\) denote the reachable states of \(A\) from the state \(q\) with respect to a transition relation \(\delta\):

\[\Sigma_0^Q = \{q' \in Q : (q, q') \in \delta^*\}\]

where \(\delta^*\) denotes the transitive closure of \(\delta\).

Definition 5.1 (Deadlock). A state \(q = (A, I, M)\) is deadlocked with respect to a transition relation \(\delta\) if there are no enabled transitions and there are actions left to be issued or matched:

\[\text{Dead}_\delta(q) \iff (I \neq A \lor M \neq A) \land \exists q' \in Q, (q, q') \not\in \delta\]

The deadlock discovery problem, given in Definition 5.2, asks whether \(F(A)\) can reach a deadlock state. This problem is NP-Complete and can be directly encoded as a propositional formula [10].

Definition 5.2 (Deadlock discovery problem).

\[\exists q \in \Sigma_0^Q : \text{Dead}_\delta(q)\]

The search for an arbitrary feasible deadlock state can be extremely expensive for many programs. We can get the search a kind of head start by finding a simple way to characterize the types of states that may deadlock. A convenient way to describe a state is by its control point. The control point of a state is simply the set of last issued actions from each process:

\[\text{Ctrl}(A, I, M) = \{a \in I : \forall a' \in I_{\text{pid}(a)}, a' \leq_{\text{po}} a, a' \in \text{req}(a')\}\]

If we only provide the last issued action for a subset of process, we obtain a partial control point that describes the collection of states which include it as a subset of their control points. Definition 5.3 augments the problem statement in Definition 5.2 to ask for a deadlock state that matches a partial program point \(D\) (also called a deadlock candidate).

Definition 5.3 (Constrained deadlock discovery problem).

\[\exists q \in \Sigma_0^Q, D \subseteq \text{Ctrl}(q) \land \text{Dead}_\delta(q)\]

6 DEPENDENCY GRAPH

This section presents a technique for generating a sound set of deadlock candidates \(D(A)\) for \(A\). The soundness property according to Definition 5.3 is formally stated in the following theorem.

Theorem 6.1 (Deadlock candidates sound). For all states \(q \in \Sigma_0^Q\). If \(\text{Dead}_\delta(q)\), then \(\exists d \in D(A), d \subseteq \text{Ctrl}(q)\).

We generate \(D(A)\) by detecting cycles in a graph \((N, E)\) where \(N = A \cup \{L_p : p \in P(A)\}\) is the set of nodes and \(E : N \times N\) is the set of edges. The node \(L_p\) is used to explicitly represent the end of process \(p\) in the graph. An edge \((a, a') \in E\) represents a potential communication dependency of \(a'\) on \(a\) in some execution of \(A\). A proof that our technique satisfies Theorem 6.1, along with proofs for the other theorems stated in this paper, is given in the appendix of the full paper [2].

Before presenting the dependency graph, we describe how one of its cycles can represent a deadlock. This not only motivates the rules for adding edges to the graph, but also leads to a precise understanding of the type of cycle the analysis must report and the types it can ignore. This is important because the graphs we
We will use the following definition to translate these concepts to a Predictive Analysis for Detecting Deadlock in MPI Programs

We will define a few new terms that allow us to talk about why a process block from progressing in the state

First, we call the deadlock action for p. Next, let a’ be the earliest action in p that is issued in q but not matched with a’ ≤<mo>mo a. We call a’ the orphaned action for p. Finally let a” be an action that is not issued in q but would allow p to progress if it is matched with a. We call a” the parent action. If the parent action does not exist, it is represented in the graph by a ⊥ node (discussed more below).

We will use the following definition to translate these concepts to the context of a path of edges in E.

Definition 6.2 (Deadlock path). Let (a₀, a₁), . . . , (aₙ₋₁, aₙ) be a path of edges in E. This path is a deadlock path for the process p ∈ P(A) if

1. pid(a₀) ≠ p and
2. pid(aᵢ) = p for all i ∈ {1 . . . n} and
3. for some i ∈ {1 . . . n − 1}, aᵢ ∈ W(A) ∪ B(A).

In a deadlock path (a₀, a₁), . . . , (aₙ₋₁, aₙ) for the process p, a₀ and a₁ are interpreted as parent and orphan actions of p. The edge connecting them represents the possibility that a₁ may depend on a₀ being issued and available to match to complete in some execution. The earliest blocking action issued by p and contained in the path is interpreted as the deadlock action of p.

Definition 6.2 requires that the deadlock action not be the last action in the path. This is because a deadlock cycle is constructed by composing the deadlock paths of two or more processes. In other words aᵢ in the deadlock path of p will be the parent action in a different deadlock path for some process p’ ≠ p. The deadlock action cannot also be a parent action because the deadlock action is issued and ready to match by definition.

Definition 6.3 (Deadlock cycle). A cycle of edges in E is a deadlock cycle if it can be constructed from a set of deadlock paths where each process in the program contributes at most one path. Additionally, the orphaned action of each deadlock path must not be a compatible match for the orphaned action of any other deadlock path in the cycle. Otherwise, the potential deadlock state would quickly unwind by matching the two issued orphaned actions.

The candidate of a deadlock cycle is the set of deadlock actions from its paths. We construct D(A) by extracting the candidate from each deadlock cycle in the graph. The proof of Theorem 6.1, referred to the full paper [2], shows that every deadlock state, there is a corresponding cycle in E that conforms to Definition 6.3 and yields a matching partial control point. The first step to constructing E is to know which actions can match together in an execution of A.

Definition 6.4 (Potential matches). For an action a ∈ A, Μ(a) denotes the set of actions that can be matched with a in some transition between reachable states of A. More precisely, if M(q) denotes the matched set in the state q, then

\[ Μ(a) = \{ a’ : \exists q, q’ ∈ Σ^*_A, q → q’ ∧ \{ a, a’ \} ⊆ M(q') \setminus M(q) \} \setminus \{ a \}. \]

Note that fully determining Μ(a) is as hard as the deadlock discovery problem itself. Fortunately, Μ(a) must only be over approximated to ensure that the whole analysis is sound. A simple over approximation would include every compatible match of the action a in Μ(a). However, the analysis presented in this paper uses the more precise approximation given in [15] which results in fewer spurious cycles.

Definition 6.5 (Edges). An edge (a, ⊥ - pid(a)) is added to E for every a ∈ A. Other edges are added between two actions a, a’ ∈ A according to the following rules:

1. If a <<mo>mo a’ then (a, a’) ∈ E.
2. If a’ ∈ Μ(a), then (a, a’), (a’, a) ∈ E.
3. If a, a’ ∈ R(A) such that a ⪯ p₀, a’ ∧ src(a) = ∗ ∧ src(a’) ≠ ∗, then (⊥ - src(a’), a’) ∈ E.
4. If a ∈ S(A) and a’ ∈ R(A) such that dst(a) = dst(a’), src(a’) = ∗, then (⊥ - dst(a), a) ∈ E.

Rule one of Definition 6.5 is an obvious preservation of the semantics presented in Section 4. It is necessary for connecting deadlock actions to other actions in the tail of a deadlock path. It also ensures that wait actions (often a deadlock action) have an incoming edge from their associated message request (often an orphaned action).

The rest of the rules ensure that parent actions are connected to orphan actions in any scenario. The most obvious occurs in rule two when the actions can form a match. The edge is bidirectional because we don’t know which action will be the orphaned action and which will be the parent action in a given deadlock.

Rules three and four add edges to encode the possibility of message starvation. Message starvation occurs when wildcard receive actions are matched with send actions in an unintended way that leaves subsequent send and receive actions without any potential future matches. In this type of deadlock the parent action does not exist in the graph (the orphaned action is waiting for an action that will never be issued). The added ⊥ node at the end of every process takes the place of the parent action in these cases.

Note that we do not need a rule for adding an edge from ⊥ nodes to wildcard receive actions because if a wildcard receive action is starved, then the resulting deadlock would have been deterministic. There is no need for an analysis in this case, as the deadlock occurs no matter how you schedule the program.

These rules will inevitably generate graphs with many spurious cycles that obviously do not correspond to real deadlock states. For example, the second rule alone forms a trivial cycle between two potentially matching actions. We only want to enumerate cycles that match Definition 6.3.

By default, Johnson’s algorithm for enumerating the elementary cycles of a directed graph [18] will enumerate all of the spurious cycles. Algorithm 1 presents a simple boolean function that can be added to Johnson’s algorithm to ensure only deadlock cycles are visited.

Johnson’s algorithm searches for cycles in one strongly connected component at a time according to a user-defined order. We extend ⪯<mo>p₀ to a total order over the nodes in the graph. The component containing the least node s is searched first and s is visited first. The global variable stack maintains a normal depth first search stack of visited nodes.
Algorithm 1 Determine whether the edge \((v, w)\) can possibly reach a deadlock cycle starting at \(s\)

1: procedure DeadlockEdge\((v, w, s)\)  
2: if \(\text{pid}(v) \equiv \text{pid}(w)\) then  
3: return true  
4: if block_count\((\text{stack}, \text{pid}(v))\) = 0 then  
5: return false  
6: if block_count\((\text{stack}, \text{pid}(v))\) = 1 \&\& \(v \in W(A) \lor B(A)\) then  
7: return false  
8: if \(\exists a \in \text{orphaned} (\text{stack}), \text{can_match}(a, w)\) then  
9: return false  
10: if \(w \neq s\) then  
11: return \(\forall a \in \text{stack}, \text{pid}(a) \neq \text{pid}(w)\)  
12: return false

The result of the chosen order is that \(s\) is always the orphaned action of the first deadlock path visited by the algorithm. Our specialized depth first search visits the successor \(w\) of the current node \(v\) when DeadlockEdge\((v, w, s)\) returns true. This ensures that the stack is always extending to a valid deadlock path according to Definition 6.2.

The block_count function returns the number of blocking actions in a given process that have been visited by the current stack. The can_match function determines whether two actions may form a compatible match based on their types and endpoints. Finally, the orphaned function returns the orphaned action from each deadlock path in the current stack.

Lines 2-3 of Algorithm 1 allow the tail of the current deadlock path on the stack to be extended along the same process (rule one in Definition 6.2). If the condition on line 2 is false, then \(v\) and \(w\) are the parent and orphaned actions for a new deadlock path. Lines 4-7 ensure that the current deadlock path contains a deadlock action that is not also a parent action for \(w\) (rules two and three in Definition 6.2). Lines 8-9 ensure that orphaned actions cannot match other orphaned actions (Definition 6.3). Lines 10-11 ensure that each process only contributes one path to the cycle (also Definition 6.3). The final case occurs when \(w\) is equal to \(s\) and a cycle is formed.

In our implementation, we also ensure that duplicate deadlock candidates are not reported by keeping track of how actions are orphaned. If the tail of a long deadlock path can reach an orphan action in another process from two different parent actions, it is simple to only enumerate one such deadlock cycle. This is left out of Algorithm 1 for clarity.

7 ABSTRACT SEMANTICS OF CONCURRENT TRACE PROGRAMS

This section defines an abstract machine \(\overline{\mathcal{F}}(A) = (Q, q_0, \rightarrow_{abs})\) that augments the semantics of CTPs to efficiently filter away infeasible deadlock candidates. This filtering is an important stage in the analysis because it can drastically reduce the number of calls to the SMT solver. The abstract transition relation \(\rightarrow_{abs}\) is shown in Figure 6 with the barrier transition omitted as it is unchanged from \(\rightarrow\). In this transition relation we create a dedicated wildcard endpoint for each source process. This eliminates the possibility of message starvation.

The Issue-Send transition generates a fresh wildcard send for the new endpoint and issues it alongside the original send action. A wait on the original send action is allowed to match in the Match-Wait transition if either the original send or the wildcard send has been matched. The wildcard send is only allowed to match wildcard receive actions issued by the destination process and the original send action is only allowed to match deterministic receive actions.

We filter a deadlock candidate \(D\) that contains the actions in \(D\), the actions process ordered before actions in \(D\) and all of the actions in other processes:

\[
A_D = \bigcup_{a \in D} \leq_{po}^{-1} [a] \cup \bigcup_{p \in \mathcal{P}(D)} A_p
\]

We then attempt to execute \(\mathcal{F}(A_D)\) to determine whether

\[
\exists q \in \Sigma_{abs}, D \subseteq \text{Ctrl}(q) \land \text{Dead}_{abs}(q)
\]

Note that this is just Definition 5.3 with the abstract transition relation substituted in. If the abstract execution is able to issue every action in \(D\), then the candidate may represent a real deadlock and it is added to the set of candidates to be encoded as an SMT formula. Otherwise, the candidate is infeasible and is discarded.
**Match order**

\[
\bigwedge_{a \in A_D} \bigwedge_{a' \in <_{ma}[a]} (c_a \implies c_{a'} \land t_{a'} < t_a)
\]

**Queue Order**

\[
\bigwedge_{a \in S(A_D) \cup R(A_D)} \bigwedge_{a' \in <_{qo}[a]} m_{a'} < m_a
\]

**Barriers**

\[
\bigwedge_{a \in B(A_D)} a' \in B_{prx}(a)(A_D) \land t_a = t_{a'}
\]

**Matches**

\[
\bigwedge_{a \in S(A_D) \cup R(A_D)} (c_a \implies \bigvee_{a' \in M_D(a) \setminus O} \text{MATCH}(a, a'))
\]

**Reach**

\[
\bigwedge_{a \in D} c_{a'}
\]

**Deadlock**

\[
\bigwedge_{a \in D \cap O} \neg c_a
\]

**No matches**

\[
\bigwedge_{a \in O} a' \in M_D(a)
\]

---

**Figure 7:** Constraints in the formula \( F \)

---

The abstract machine is sound if it never discards a reachable control point. Let the set of reachable control points from a state \( q \) and a transition relation \( \delta \) be \( C^q_\delta \).

\[
C^q_\delta = \{ \text{Ctrl}(q') : q' \in S^q_\delta \}
\]

Theorem 7.1 states that the reachable control points of the abstract machine subsume the reachable control points of the concrete machine. Theorem 7.2 states that if the abstract machine cannot issue every action in the deadlock candidate in one execution, it will not be able to issue them all in any execution. Together these theorems prove that the candidate \( D \) can be filtered away in a single execution of the abstract machine when it fails to issue all of the actions in \( D \).

**Theorem 7.1 (Abstract candidate simulation).** Let \( q \in S^q_\delta \) and \( q' \in \Sigma_{\text{abs}}^q \) be a concrete and abstract state reachable from the start state \( q_0 \). If \( \text{Ctrl}(q) = \text{Ctrl}(q') \), then for all control points \( D \in C^q_\delta \), it follows that \( D \in C^{q'}_{\text{abs}} \).

**Theorem 7.2 (Abstract deadlock deterministic).** Let \( q \in S^q_{\text{abs}} \) be a reachable abstract state with \( \text{Dead}^q_{\text{abs}}(q) \). If \( D \not\subseteq \text{Ctrl}(q) \), then for all \( q' \in S^{q'}_{\text{abs}} \), \( D \not\subseteq \text{Ctrl}(q') \).

---

**8 SMT encoding**

If \( \overline{F}(A_D) \) is able to issue each action in the candidate \( D \), then it is used to construct an SMT formula \( F \). A satisfying assignment for \( F \) can be used to construct a witness execution for the deadlock candidate. If \( F \) is unsatisfiable, then there is no feasible deadlock state that contains \( D \) as part of its control point. The encoding is an extension of the one designed by Huang et al. [16] for finding zero-buffer compatible executions of MPI programs.

Let \( M_D \) be an over approximated set of match pairs for \( A_D \). For each action \( a \in A_D \), the encoding creates an integer variable \( t_a \) to hold its completion timestamp in the witness execution. If \( a \in S(A_D) \cup R(A_D) \), then the encoding also creates an integer variable \( m_a \) to reference the timestamp of the action that matches \( a \). Additionally, \( w_a \) is used as another name for \( t_w \) where \( w \in W(A_D) \) and \( a = \text{req}(w) \). The new encoding adds a boolean variable \( c_a \) for every action that is true if \( a \) must be issued and completed in the witness execution.

The rules for the encoding are shown in Figure 7. The Match order and Queue order constraints preserve the meaning of \( <_{\text{mo}} \) and \( <_{\text{qo}} \) in the encoding. An action can only complete if its \( <_{\text{mo}} \) predecessors have completed and the timestamps must reflect that. Additionally, matches must conform to the non-overtaking guarantee of MPI executions. In all cases, constraints are omitted when they are obviously redundant with respect to existing constraints and the transitivity of \(<\) and \(=\) over the integers.

The Barriers constraint encodes the inter-process synchronization behavior of barrier actions by asserting that groups complete at the same time. All other timestamps, including \( m_a \) for each \( a \in A_D \), are asserted to be distinct.

The Matches constraint forces send and receive actions to find a match if they must complete in the witness execution. Some matches are not allowed assuming the execution ends in a deadlock parked at the actions in \( D \). Specifically, let \( O \) denote the set of orphaned send and receive actions for the candidate \( D \).

\[
O = \{ a \in A_D : \exists a' \in W(A_D) \cap D, a = \text{req}(a') \}
\]

The Matches constraint purposely excludes the actions in \( O \) as they should not be matched in the witness execution.

In Section 4, a send and receive action were matched by copying them from the issued set into the matched set together. After that, their wait actions were allowed to complete. This semantic meaning is preserved in the encoding by the MATCH \((a, a') \) constraint which expands to

\[
c_{a'} \land t_{a'} = m_{a'} \land t_a = m_a \land t_w < w_{a'} \land t_{a'} < w_a.
\]

First, this constraint records that \( a' \) is completed by the match. Second, the actions complete together by recording the timestamp of the other in the \( m \) variables. Finally, both timestamps are required to precede the timestamps of both wait actions. Note that in the infinite-buffer setting, the wait for the send action does not exist and so one of these constraints is omitted.

The Reach constraint asserts that every predecessor of the actions in \( D \) is completed except the actions in \( O \). In other words, it asserts that the deadlock \( D \) is reachable.

The Deadlock constraint simply asserts that the deadlock actions in \( D \) and the orphaned actions in \( O \) are not complete. The No matches constraint ensures that any issued actions that could untangle the deadlock are complete, thus forcing them to find matches that exclude the deadlock and orphaned actions. Given a satisfying assignment of the variables in \( F \), the witness execution can be constructed from the \( t \) timestamp variables while the matches made can be recovered by consulting the \( m \) variables.

---

**9 EXPERIMENTS**

The runtime trace for our approach is observed through code instrumentation. The MPICH library is used for the actual runtime [23]. The translation of an observed MPI execution trace to a CTP is
Table 1: Tests on Selected Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>#Procs</th>
<th>#Calls</th>
<th>B</th>
<th>D</th>
<th>Tm</th>
<th>ISP Tm</th>
<th>MOPPER Tm</th>
<th>MOPPER-o Tm</th>
<th>Aislinn Tm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>35</td>
<td>0</td>
<td>√</td>
<td>0.037s</td>
<td>0.223s</td>
<td>0.244s</td>
<td>0.258s</td>
<td>1.665s</td>
</tr>
<tr>
<td></td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td>0.002s</td>
<td>0.957s</td>
<td>0.496s</td>
<td>0.488s</td>
<td>1.672s</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>75</td>
<td>0</td>
<td>√</td>
<td>0.055s</td>
<td>TO</td>
<td>0.503s</td>
<td>0.619s</td>
<td>186.190s</td>
</tr>
<tr>
<td></td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td>0.007s</td>
<td>TO</td>
<td>1.357s</td>
<td>1.870s</td>
<td>175.042s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>155</td>
<td>0</td>
<td>√</td>
<td>0.206s</td>
<td>TO</td>
<td>2.406s</td>
<td>2.302s</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td>0.052s</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>Integrate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>36</td>
<td>∞</td>
<td></td>
<td>&gt;1000s</td>
<td>0.328s</td>
<td>0.303s</td>
<td>0.383s</td>
<td>3.832s</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>46</td>
<td>∞</td>
<td></td>
<td>TO</td>
<td>23.186s</td>
<td>0.392s</td>
<td>16.028s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>76</td>
<td>∞</td>
<td></td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td>Diffusion2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>52</td>
<td>0</td>
<td>√</td>
<td>0.325s</td>
<td>TO</td>
<td>N/A</td>
<td>N/A</td>
<td>1.245s</td>
</tr>
<tr>
<td></td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td>0.049s</td>
<td>32.005s</td>
<td>0.521s</td>
<td>0.486s</td>
<td>1.232s</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>108</td>
<td>0</td>
<td>√</td>
<td>0.761s</td>
<td>TO</td>
<td>N/A</td>
<td>N/A</td>
<td>135.273s</td>
</tr>
<tr>
<td></td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td>0.06s</td>
<td>TO</td>
<td>0.792s</td>
<td>86.112s</td>
<td></td>
</tr>
<tr>
<td>Floyd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>120</td>
<td>∞</td>
<td></td>
<td>0.071s</td>
<td>TO</td>
<td>3.472s</td>
<td>0.623s</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>256</td>
<td>∞</td>
<td></td>
<td>0.133s</td>
<td>TO</td>
<td>24.814s</td>
<td>1.256s</td>
<td>TO</td>
</tr>
<tr>
<td>IS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>56</td>
<td>∞</td>
<td></td>
<td>0.001s</td>
<td>1.054s</td>
<td>1.172s</td>
<td>1.809s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>120</td>
<td>∞</td>
<td></td>
<td>0.004s</td>
<td>1.426s</td>
<td>3.972s</td>
<td>5.139s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>312</td>
<td>∞</td>
<td></td>
<td>0.457s</td>
<td>TO</td>
<td>3.092s</td>
<td>2.205s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>632</td>
<td>∞</td>
<td></td>
<td>1.559s</td>
<td>TO</td>
<td>5.331s</td>
<td>5.068s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>828</td>
<td>0</td>
<td></td>
<td>0.03s</td>
<td>45.471s</td>
<td>0.621s</td>
<td>0.599s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.01s</td>
<td>46.565s</td>
<td>0.589s</td>
<td>0.602s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1660</td>
<td>0</td>
<td></td>
<td>0.014s</td>
<td>148.251s</td>
<td>1.394s</td>
<td>1.530s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.024s</td>
<td>150.570s</td>
<td>1.767s</td>
<td>1.539s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>3224</td>
<td>0</td>
<td></td>
<td>0.036s</td>
<td>561.294s</td>
<td>8.328s</td>
<td>7.731s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.07s</td>
<td>476.516s</td>
<td>12.714s</td>
<td>4.367s</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^a\) Aislinn aborts the verification for unknown exceptions after 2 minutes of running.  
\(^b\) Aislinn does not support some MPI calls in the program.  
\(^c\) MOPPER is not launched because the process of trace generation finds a deadlock.

9.1 Results

The experiments compare the performance of our approach with three state-of-the-art MPI verifiers MOPPER, a SAT based tool [10, 11], ISP, a dynamic analyzer [25, 30], and Aislinn, another dynamic analyzer [4]. MOPPER is a trace based verifier that checks the same behaviors described in this paper. Therefore, the comparison to our approach is direct. The experiments compare the results of two versions of MOPPER, the original tool [11] and the optimized tool [10]. ISP is a general verification tool that is not specialized to only deadlock but includes races and user defined assertions. The comparison to our approach is intended to convey the benefit of specialization to just deadlock and suggest that similar gains may be found in specializing to just data race or assertions. Aislinn is a dynamic verifier that covers two buffering choices for each send operation in a program. In other words, Aislinn detects more behaviors than those in the infinite buffer semantics (buffering provided for all sends) and the zero buffer semantics (no buffer for any send). In order to compare to our approach, we set the buffering mode for Aislinn to “eager” (equivalent to the infinite buffer setting) and “rendezvous” (equivalent to the zero buffer setting).

The results of the comparison are in Table 1. The column “B” represents the buffering setting in the runtime. The column “D” indicates the existence of deadlocks. The column “Tm” is the time of running our approach and constraint solving if necessary. The “Tm” column for ISP is the running time of dynamic analysis. The “Tm” columns for MOPPER and optimized MOPPER are for constraint generation and solving. The “Tm” column for Aislinn is the running time of the tool for either “eager” mode or “rendezvous” mode. The notation “TO” means “time out” (exceeding the time limit set for each test). The notation “N/A” means “not available.”
9.2 Benchmarks

Monte [5] implements the Monte Carlo method by a master-slave model. The program has a deadlock trace under the zero buffer setting.

Integrate [1] implements the algorithm that computes an integral of the sin function by using a large number of wildcard receives to match the sends from multiple sources.

Diffusion2D [1] has an interesting computation pattern that uses barriers to “partition” the message communication into several sections. A message from a send can be only received in a common section. Deadlock occurs under the zero buffer setting.

Floyd [33] implements the shortest path algorithm for all the pairs of nodes. The message communication is only built between any two successive processes.

GE [33] is a message passing implementation for Gaussian Elimination. Messages are communicated by issuing several wildcard receives on each node.

Heat [24] implements the solution of the heat conduction equation. The communication pattern for this benchmark contains several message starvation deadlocks.

IS [3] implements the solution of integer sorting. The message passing in this benchmark is deterministic.

9.3 Discussion

The comparison of time cost to the three existing state-of-the-art tools demonstrates that our approach is much more efficient for deadlock detection in many cases. For the benchmark programs with a small number of processes (e.g., Monte with 4 processes), all the tools are able to finish the tests very quickly. However, as the number of processes grows, implying that the benchmark has a much higher degree of non-determinism, ISP and Aislinn both have unexpectedly high time costs for the tests (e.g., Floyd with 16 processes), and even time out for some tests. Overall, MOPPER is much quicker than the other two tools. The optimized MOPPER is even more efficient, especially for large, complicated benchmarks (e.g., Integrate with 16 processes). However, it still suffers from the scalability problem for a few tests (e.g., Monte with 16 processes). In contrast, our approach is able to finish most tests under a second.

9.4 Effectiveness of Filtering

Table 2 shows the effectiveness of the cycle detection algorithm and the abstract machine as a filtering mechanism. Each test shown in Table 2 is an exhaustive enumeration on the detected deadlock candidates by our approach. The test here is different from those in Table 1 as the enumeration does not terminate even if a feasible deadlock is validated. Therefore, some tests in Table 2 may take much more time to run. The times are not recorded in Table 2 as they are meaningless for actual deadlock analysis. The column “#Edges” records the number of edges in the dependency graph. The column “#Instances” records the number of deadlock candidates. The column “#Filtered” records the number of deadlock candidates filtered away by the abstract machine. The column “#Deadlock” records the number of real deadlocks.

The numbers shown in Table 2 indicate that the abstract machine in our analysis has a high precision for filtering out infeasible deadlocks. The percentage of filtered deadlock instances among all the detected instances varies for the test. But if an instance is not filtered, then it is often a real deadlock. The modified cycle detection algorithm is also effective at enumerating a small number of cycles for very large graphs with a huge number of spurious cycles.

### Table 2: Enumerating All Candidates on Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>#Procs</th>
<th>B</th>
<th>#Edges</th>
<th>#Instances</th>
<th>#Filtered</th>
<th>#Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte</td>
<td>4</td>
<td>90</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Monte</td>
<td>8</td>
<td>286</td>
<td>19</td>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Monte</td>
<td>16</td>
<td>966</td>
<td>44</td>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integrate</td>
<td>8</td>
<td>∞</td>
<td>246</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Integrate</td>
<td>10</td>
<td>∞</td>
<td>370</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Integrate</td>
<td>16</td>
<td>∞</td>
<td>886</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Diffusion2D</td>
<td>4</td>
<td>385</td>
<td>16</td>
<td>13</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Diffusion2D</td>
<td>4</td>
<td>∞</td>
<td>385</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Diffusion2D</td>
<td>8</td>
<td>1135</td>
<td>46</td>
<td>45</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Diffusion2D</td>
<td>8</td>
<td>1135</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Floyd</td>
<td>8</td>
<td>1170</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Floyd</td>
<td>16</td>
<td>2754</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GE</td>
<td>8</td>
<td>∞</td>
<td>186</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GE</td>
<td>16</td>
<td>∞</td>
<td>522</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Heat</td>
<td>16</td>
<td>∞</td>
<td>1458</td>
<td>104</td>
<td>102</td>
<td>2</td>
</tr>
<tr>
<td>Heat</td>
<td>32</td>
<td>∞</td>
<td>3458</td>
<td>454</td>
<td>452</td>
<td>2</td>
</tr>
<tr>
<td>IS</td>
<td>64</td>
<td>0</td>
<td>442</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IS</td>
<td>128</td>
<td>0</td>
<td>890</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IS</td>
<td>256</td>
<td>0</td>
<td>1786</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

10 RELATED WORK

Much of the literature on predictive program analysis focuses on detecting as many errors as possible from a single observed execution [8, 20]. As mentioned in Section 1, the analysis presented here is maximally predictive for single-path programs but cannot reason about messages that are issued in a schedule-dependent manner. Instead, this paper focuses on improving the efficiency of an SMT based analysis for single-path programs while leaving extensions to more general programs as future work.

The approach in this paper is inspired by several works. Joshi et al. proposed a method for multi-threaded Java programs by first detecting potential lock dependency cycles with an imprecise dynamic analyzer and then finding real deadlocks by a random thread scheduler with high probability [19]. The refining strategy of the work inspires the staged approach taken in this paper, but the tool presented in this paper is guaranteed to report a deadlock if it exists in the single-path program.

Sherlock is a tool that uses concolic execution for deadlock detection in Java programs [8]. The key idea is similar to our approach: finding potential deadlocks, and then searching for a feasible schedule that leads to the deadlock. The difference is that Sherlock repeatedly finds alternate schedules through solving constraints that describe new permutations of previously observed schedules rather than leveraging an abstract machine to filter out false deadlocks.
A precise SMT encoding technique was proposed by Huang et al. for verifying properties over MCAPI programs containing message race [17]. The encoding does not require a precise match set and was extended to checking zero buffer incompatibility for MPI programs [16]. This technique is adapted for validating deadlocks in this paper.

The POE approach is a dynamic partial order reduction solution [9] for MPI program verification [25, 30]. The approach was extended to POE_MSE, which first uses a precise happens-before relation to find the potential sends that may cause different behaviors based on the initial trace, then replays the execution at each potential send with a different choice, i.e. buffering the send instead of matching it [31].

MOPPER is an MPI deadlock detector based on boolean satisfiability encoding [10, 11]. While the solution is precise, the size of the encoding is cubic meaning it only scales to a low degree of message non-determinism.

CIVL is a model checker that uses symbolic execution to verify a number of safety properties of various types of concurrent programs including message passing programs [28, 29, 34]. The tool is outperformed by MOPPER.

An extension to the model checker SPIN [14], is MPI-SPIN that is specific to verifying MPI programs [26, 27]. Since a massive number of states are explored, the work is not scalable.

Böhm et al. provide an approach that aims to find deadlocks for an MPI program under both environments of synchronization and no synchronization [4]. The approach first uses standard partial order reduction to find deadlocks assuming the environment has no synchronization. It then uses an algorithm to search missed deadlocks by enforcing synchronization in the basic operations such as send and collective operations.

MPI-Checker is a static analyzer based on abstract syntax tree of the source code of MPI programs [7]. The tool is able to check many errors in a program, However, it is limited to check deadlocks caused by complicated semantics of communication.

ParTypes is a type-based approach for verifying MPI programs by developing a protocol language for a system [22]. Since the approach is able to avoid traversing the state space, the analysis is scalable for large programs.

Umpire is an approach of runtime verification for checking multiple MPI errors such as deadlock and resource tracking [32]. The error checking is taken by spawning one manager thread and several outfilder threads in the execution of an MPI program. An extension to Umpire is Marmot [21]. The work uses a centralized server instead of multiple threads for error checking. Another extension to Umpire is MUST [12, 13]. The structure of MUST allows the users to execute the error checking either in an application process itself or in extra processes that are used to offload these analyses. However, just like Umpire and Marmot, the approach is neither sound nor complete for deadlock detection.

11 CONCLUSION

This paper presents a new approach that automatically detects deadlocks in single-path MPI programs after observing a single execution. The approach leverages a simple characterization of deadlock to efficiently detect deadlock candidates in a dependency graph. An abstract machine is used to quickly disregard many infeasible candidates while the remaining candidates are precisely validated by an SMT solver with an efficient encoding for deadlock. The approach is sound and complete for deadlock detection in any single-path MPI program on a given input. Experiments show that the new approach finishes typical benchmarks within only 1 second while the other state-of-the-art MPI verifiers time out. Future work considers more filtering techniques and extending the approach to support multiple-path MPI programs with more complicated structures.

REFERENCES


