Appears in Supercomputing ´92, November 1992, Minneapolis, MN

Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs

Robert H. B. Netzer
Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912
rn@cs.brown.edu

Barton P. Miller
Computer Sciences Dept.
University of Wisconsin–Madison
1210 W. Dayton St.
Madison, WI 53706
bart@cs.wisc.edu

Abstract

A common debugging strategy involves re-executing a program (on a given input) over and over, each time gaining more information about bugs. Such techniques can fail on message-passing parallel programs. Because of variations in message latencies and process scheduling, different runs on the given input may produce different results. This non-repeatability is a serious debugging problem, since an execution cannot always be reproduced to track down bugs. This paper presents a technique for tracing and replaying message-passing programs for debugging. Our technique is *optimal* in the common case and has good performance in the worst case. By making run-time tracing decisions, we trace only a fraction of the total number of messages, gaining two orders of magnitude reduction over traditional techniques which trace every message. Experiments indicate that only 1% of the messages often need be traced. These traces are sufficient to provide replay, allowing an execution to be reproduced any number of times for debugging. Our work is novel in that we use run-time decisions to detect and trace only those messages that introduce nondeterminacy. With our strategy, large reductions in trace size allow long-running programs to be replayed that were previously unmanageable. In addition, the reduced tracing requirements alleviate tracing bottlenecks, allowing executions to be debugged with substantially lower execution-time overhead.

1. Introduction

Parallel programs can be nondeterministic. When processes communicate by passing messages, variations in scheduling and message latencies can cause two executions of the same program (on the same input) to produce different results. Such nondeterminacy may be intended, but it can cause serious problems while debugging: subsequent executions of the program may not reproduce the original bug. This non-repeatability makes it difficult to use traditional sequential debugging techniques that require repeated execution. Therefore, a mechanism for tracing and then replaying a program execution is an essential part of parallel program debugging. A critical cost in such a mechanism is the cost of tracing. In this paper we present a trace and replay mechanism for message-passing parallel programs that is *optimal* in the common case and has good performance in the worst case. We significantly reduce the number of messages that need be traced, improving by up to two orders of magnitude earlier techniques that trace every message. With such a reduction, long-running programs can now be replayed that could not have been previously replayed. Our tracing works by making decisions at run-time, tracing often the minimal set of messages needed to provide replay. Experiments show that only 1% of the messages usually need be traced.

In a trace and replay scheme, the order in which messages are delivered (but not their contents) is first traced during execution. These traces are then used during re-execution to force each message to be delivered to the same operation as during the traced execution. Tracing the original execution is necessary because some messages may *race* with others, introducing nondeterminacy into the execution. Two messages race if they are simultaneously in transit and either could arrive first at some receive operation. If the original order in which racing messages are delivered is not recorded, their order cannot always be reproduced during replay. However, by tracing the original message deliveries and then forcing them to occur during replay, the computation and all its messages will be exactly reproduced†. An erroneous execution can then be repeatedly replayed to analyze the execution more carefully and gain information about bugs.

Our main result is a tracing strategy using on-the-fly detection of racing messages that is *optimal* in most

This work was supported in part by National Science Foundation grants CCR-8815928 and CCR-9100968, Office of Naval Research grant N00014-89-J-1222, and a grant from Sequent Computer Systems Inc.

† Interactions with the external environment must also be reproduced (such as return values from system calls). However, these interactions must be reproduced to replay sequential programs as well.

cases and effective even in the worst case. Instead of tracing every message (as earlier schemes propose), our technique checks each message to determine if it races with another, and traces only one of the racing messages. When a message is received, a race check is performed by analyzing the execution order between the previous receive operation in the same process and the message sender. The ordering information necessary for this check is maintained during execution by appending vector timestamps onto user messages. This strategy is effective because the racing messages are exactly those that introduce nondeterminacy into the execution.

Our work is novel in that only the racing messages are traced. In contrast, earlier trace and replay schemes for message-passing programs require tracing every message. Replay was first introduced by Curtis and Wittie in the BugNet system for debugging distributed C programs[1]. LeBlanc and Mellor-Crummey[4] also addressed replay but considered both shared-memory and message-passing parallel programs. They trace only the order in which messages are delivered (and not their contents). By reproducing only the order of message deliveries, their contents (and hence the original computation) will also be reproduced. However, both of these schemes require emitting some type of trace for every message. Tracing every message can require huge amounts of storage for long-running programs, making them impossible to debug. In addition, as processors become faster and parallel machines become larger, tracing becomes an increasing bottleneck.

2. Example

To contrast traditional trace and replay schemes with our tracing strategy, we present an example message-passing program. This example shows that tracing every message sent during execution is sufficient to provide replay but is not necessary. Instead, tracing only the racing messages is sufficient to provide correct replay of all messages (even those that do not race).

Figure 1a shows a three-process message-passing program. Processes P_1 and P_3 send `Msg1` and `Msg2` to process P_2. Process P_2 issues two `Recv` operations that will accept messages from any process. Figure 1b illustrates one possible execution of this program in which P_2 first received `Msg1` sent by P_1, then received `Msg2` from P_3. However, because these two messages *race*, they are not guaranteed to be delivered in this order. Intuitively, two messages race if either could be received first (due to the unpredictability of schedulers and message delays). For example, if `Msg1` were delayed (because of variations in message latencies), `Msg2` could instead be received first by P_2, as shown in Figure 1c. This nondeterminacy causes a problem when debugging, since re-executing the program (on the same input) is not guaranteed to reproduce the original execution.

To replay the execution for debugging, we must first trace the order in which the messages are delivered, and then use this trace to force a re-execution to exhibit the same message deliveries. Earlier trace and replay schemes propose tracing the order in which *all* messages are delivered[1, 4]. For example, they would record that `Msg1` was delivered to the first `Recv` in P_2 and that `Msg2` was delivered to the second `Recv`. During replay, the receive operations are modified to accept only the appropriate messages. However, in this example, it suffices to trace only one of the two messages. If only the delivery of `Msg1` to the first `Recv` in P_2 is recorded, sufficient information still exists for replay. By forcing *only* this message to be delivered to the appropriate receive (the first `Recv` in P_2), the other message will automatically be delivered to the correct operation — it has no where else to go. One of our results is a proof that only racing messages need be traced, and we need trace only one message in each race. Non-racing messages cannot introduce nondeterminacy and thus their deliveries need not be enforced during replay.

In Section 3 we formally define these races. In Section 4 we show how to detect and trace them on-the-fly and provide replay from the traces. We also prove that in the common case our tracing strategy is optimal: the traced messages are often the minimal number whose order must be reproduced for replay. In Section 5 we present experimental results indicating that this strategy is effective in practice, even in the non-optimal case.

3. Formal Characterization of Races

In this section we formally define the notion of a message that races. In the following sections we show how to detect racing messages on-the-fly to record information sufficient for replay. To formalize the races in a program execution, we present a model that represents both the actual behavior exhibited by the execution (such as its events and message deliveries) and behaviors it potentially could have exhibited. We then formalize the notion that a race exists when either of two messages could have been delivered first to some receive operation.

3.1. Intuitive Example

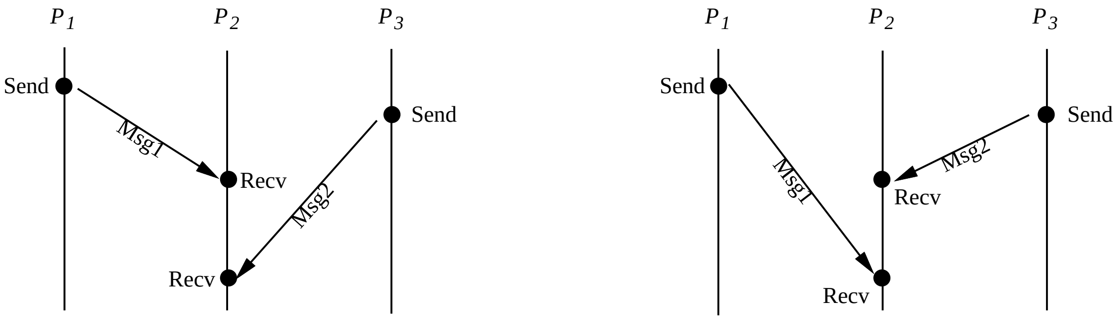
Intuitively, two messages race if either *could have* been accepted first by some receive, due to variations in messages latencies or process scheduling. To isolate the effects of these variations, we will examine all possible ways in which a *frontier* can be drawn across the execution. A frontier divides the events into two sets: those before the frontier and those after. Figure 2a shows an example frontier just above *a*, *b*, and *c*. We are interested in all frontiers such that

(1) two (or more) sends are just after the frontier (events *a* and *c* in Figure 2),

P_1 P_2 P_3

```
Send Msg1 to P2      Recv X from ANY      Send Msg2 to P2
                     Recv Y from ANY
```

(a)



(b) **(c)**

Figure 1. (a) example message-passing program, and (b),(c) two possible executions

(2) a receive that could have accepted either of their messages is just after the frontier (event *b* in Figure 2), and

(3) all receive events before the frontier also have their senders before the frontier.

All such frontiers identify receive operations (those just after the frontier) that could have accepted different messages. We say that messages sent to such receives are involved in *frontier races*. As shown in Figure 2b, the race between the messages below the frontier means that a different message (from *c*) could have arrived at *b* first, forcing the message from *a* to be received later.

3.2. Program Execution Model

We next formally characterize the frontier races in terms of a model. Our model is simply a notation for representing the execution of a message-passing parallel program. A *program execution* is a pair, $P = \langle E, \xrightarrow{HB} \rangle$, where E is a finite set of *events* and $\xrightarrow{HB}$ is the *happened-before relation* defined over E.

An event represents the execution instance of a set of consecutively executed statements in some process. We assume a fixed number of processes, p, exist during execution, and denote the events in process p by E_p. We distinguish between *synchronization* events, which are instances of send or receive operations, and *computation* events, which represent the execution of non-synchronization statements. We model message passing as occurring over *logical channels*; each send or receive event e specifies a set of logical channels over which it sends or is willing to receive a message. Using logical channels is very general; any message-passing scheme (such as ports, mailboxes, or links) can be represented. We attach the attributes $SEND(e)$ and $RECEIVE(e)$ to synchronization events to denote these channels.

The happened-before relation, $\xrightarrow{HB}$, shows how events potentially affect one another[3], and is defined as the irreflexive transitive closure of the union of two other relations: $\xrightarrow{HB} = (\xrightarrow{XO} \cup \xrightarrow{M})^+$. The $\xrightarrow{XO}$ relation shows the order in which events in the same process execute. The i^{th} event in any process p (denoted $e_{p,i}$) always executes before the $i+1^{\text{st}}$ event: $e_{p,i} \xrightarrow{XO} e_{p,i+1}$. The $\xrightarrow{M}$ relation shows the order in which messages are delivered: $a \xrightarrow{M} b$ means that a sent a message that b received (we also write $a \xrightarrow{M} b$ to denote the message a sent). An event a is said to happen before an event b iff a could affect b because they belong to the same process or because a sequence of messages was sent from a (or a following event) to b (or a preceding event).

3.3. Definition of Race

As illustrated above, a frontier race exists when a receive event b could receive a message from one of several sends. To formalize these races, we need to focus on other executions that could have occurred, which we call *feasible program executions*. We will characterize the set of all feasible executions that show all the ways in which a frontier can be drawn across the execution. We

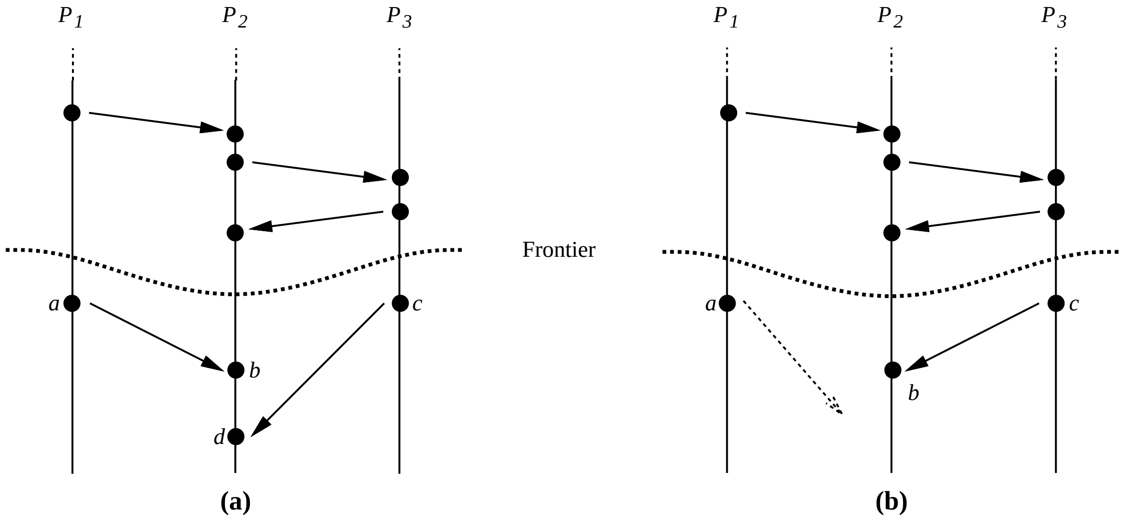


Figure 2. (a) one possible frontier, (b) the message from *c* could have been received by *b*

can then define the frontier races in terms of this set.

For each frontier that can be drawn across the execution, P, we will consider the set of all feasible executions that exhibit the same events and message deliveries as P up to that frontier, after which different messages deliveries may occur. We denote this set by F_{SAME}.

Definition 3.1

F_{SAME} is the set of all program executions, $P' = \langle E', \xrightarrow{HB'} \rangle$, such that the following hold.

(1) P' represents an execution that the program could actually perform.
(2) P' performs the same events as P to just after some frontier, defined as follows:
for each process p, the set $E_p{}'$ includes
(a) a prefix of the events in E_p (which defines the events before the frontier), and
(b) the event in E_p just after this prefix (which defines the event after the frontier).
(3) P' exhibits the same message deliveries as P before the frontier:
for all $a,b \in E'$ where a and b are before the frontier, $a \xrightarrow{M'} b \Leftrightarrow a \xrightarrow{M} b$. ■

Figure 2 illustrates an actual program execution (P) and a feasible program execution (P') in F_{SAME}. P' (Figure 2b) exhibits the identical events and message deliveries as P (Figure 2a) before the frontier, and includes one event in each process just after the frontier. Although the messages before the frontier must be sent to the same events in P' as in P, the messages involving events just after the frontier may have different destinations. This example shows that even though a message was delivered from a to b in the actual execution (P), a different execution was possible (P') in which a message was delivered from c to b and the message from a was delivered to some later event.

Since F_{SAME} characterizes message deliveries other than those that actually occurred, we can formally define a frontier race in terms of this set. We will say that two messages, $a \xrightarrow{M} b$ and $c \xrightarrow{M} d$, are involved in a *frontier race* iff a frontier can be drawn in such a way that a, b, and c are just after the frontier, and b could have received the message sent by c. We define a binary relation (over the messages in P) to represent these races.

Definition 3.2

$a \xrightarrow{M} b$ *RacesWith* $c \xrightarrow{M} d$ iff a program execution, $P' = \langle E', \xrightarrow{HB'} \rangle$, exists in F_{SAME} such that $a,b,c \in E'$ and $c \xrightarrow{M'} b$ $(c \neq a)$. ■

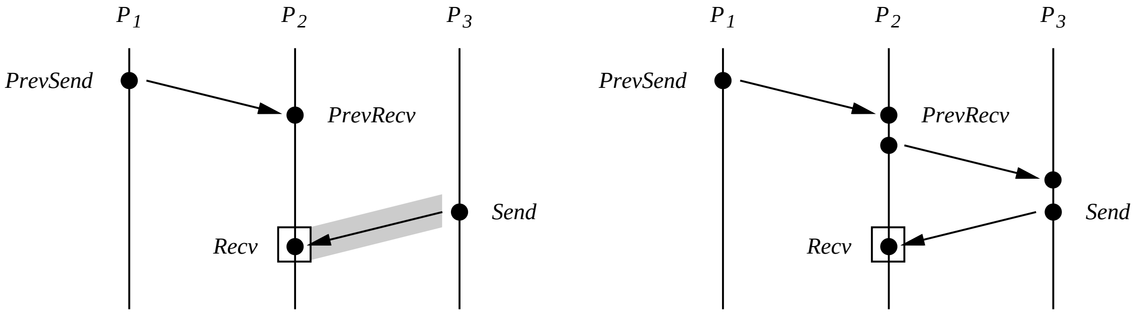
4. Message Tracing Using On-the-fly Race Detection

We now present our trace and replay strategy. Our approach is to locate the frontier races on-the-fly and to trace only one message in each race, instead of tracing every message. In this section we first show how the frontier races can be detected and traced on-the-fly, and then discuss how to provide replay from the traces. We also prove that our algorithm is optimal in most cases and traces only the minimal number of messages required.

Receive *Msg* **from** *Channels*:

1: *Send* = event that sent *Msg*;
2: *PrevRecv* = previous event (in the same process) willing to receive from the channel over which *Msg* was sent;
3: *PrevSend* = event that sent message to *PrevRecv*;
4: **if** (*PrevRecv* $\not\xrightarrow{\text{HB}}$ *Send*)
5: trace that a message was delivered from *Send* to *Recv*;

(a)



(b) Frontier race (shaded message traced)
(*PrevRecv* $\not\xrightarrow{\text{HB}}$ *Send*)

(c) No frontier race (no messages traced)
(*PrevRecv* $\xrightarrow{\text{HB}}$ *Send*)

Figure 3. (a) tracing algorithm, (b),(c) example race checks performed at boxed receive

4.1. On-the-fly Race Detection and Tracing

We detect frontier races on-the-fly by performing a race check after each receive. By analyzing the execution order between the sender and a previous receive in the same process, we can determine whether the received message races with another, and trace only a racing message. For simplicity, we assume that the receiving ends of logical channels are associated with a *single* process; e.g., messages to ports (but not mailboxes). Two messages can then race only if they are received by the same process, simplifying the tracing algorithm. Below we discuss handling more general (mailbox) communication.

Figure 3a shows our on-the-fly race detection and tracing algorithm, which is invoked after each receive. Recall that a race exists when either of two messages could have arrived first at some receive. After a message is received, this algorithm determines whether the message could have instead been received by a previous operation in the same process. To identify these situations, an earlier receive is located (line 2) that accepted a message from the same channel over which the current message was sent. Both the earlier message and the current message are race candidates. As shown in Figure 3b, if *PrevRecv* did not happen before the sender of the current message (*Send*), then a frontier race exists: both the previous and current messages could have been simultaneously in transit and either could have arrived first at *PrevRecv*. In this case the algorithm traces the second racing message. If instead *PrevRecv* happened before *Send* (as shown in Figure 3c), then no race exists: the two messages could not have been simultaneously in transit, and the algorithm emits no trace. We prove in Appendix A (Theorem 1) that this algorithm traces at least one message in each frontier race.

The traces only need identify the sending and receiving events of the racing message. These events can be identified by maintaining in each process a local counter (incremented after every synchronization operation) that is used to assign serial numbers to events[4]. It suffices to trace the event serial numbers of the sender and receiver and the process number of the sender. If one trace file is maintained for each process in the program execution, the process number of the receiver is implicit and need not be recorded.

Because we assume that the receiving end of each logical channel is associated with a single process, to find races it suffices (in line 2) to locate the previous event in the *same* process that could have accepted the incoming message. In mailbox communication, a mailbox might have multiple simultaneous owners in different processes. Two messages (to the same mailbox) can race even if they are received by different processes. Detecting these races requires locating earlier events in any process that could have accepted the incoming message sent to the mailbox. Such events can be located by modifying the mailbox mechanism to store the last event in each process that received a message from the mailbox. Line 4 of the tracing algorithm can be modified to check the execution of order of each of these events against the sender.

4.2. Replay

During replay each frontier race must be resolved in the same way as during the original execution. We prove in Appendix A (Theorem 2) that the delivery of only the traced messages need be specially enforced during replay; untraced messages will automatically arrive at the correct receive and need no special treatment. We provide replay by tagging and buffering the racing messages so they can be accepted by receive operations in the proper order. If necessary, the traced messages can be forced (at additional cost) to also arrive in their original order, alleviating the need for buffering and ensuring that no buffer overflows occur during replay.

To effect replay, the trace files must first be collated; a racing message is traced during execution when it is *received*, but tagging the message during replay requires a special action when it is *sent*. To produce the collated trace file for process p, all traces of messages sent from p must be collected (from the uncollated trace files) into a single file and sorted by sender serial number. During replay, as in the original execution, serial numbers must be assigned to events by maintaining a local counter that is incremented after every synchronization operation. This counter is used to ensure that racing messages are accepted by the intended receive. Before each send, the serial number of the next racing message (read from the trace file) is compared to the current value of the local counter. Because the trace file is sorted, these numbers will match if the message about to be sent was originally involved in a race. If the message to be sent originally raced, it is tagged with the serial number of its intended receive. If the message did not race, it is not specially tagged. Receives are modified to accept tagged messages only if their serial numbers match those on the tagged messages. Tagged messages with serial numbers that do not match are buffered so they can be accepted by later receives. Such buffering is often normally performed by message-passing systems that accept asynchronous messages. Untagged messages are accepted as usual.

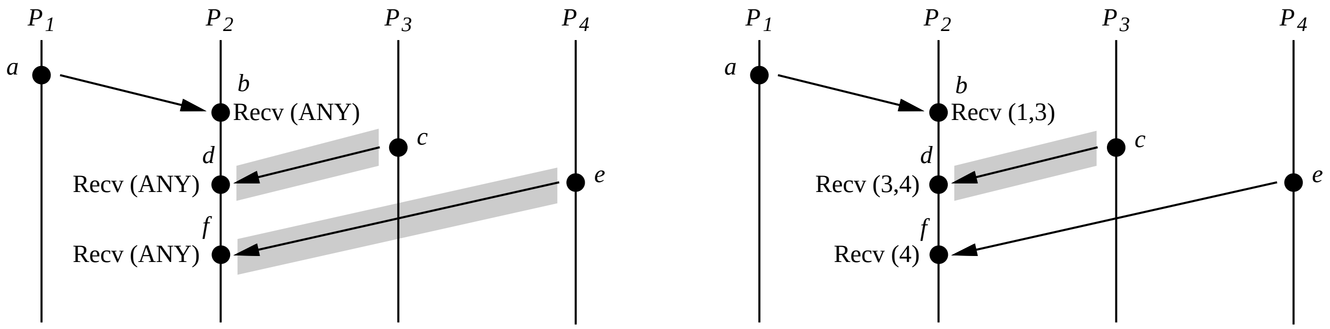
The above replay strategy ensures that each message is *received* by the correct operation, but does not guarantee that messages *arrive* at a process in the same order as during the original execution. Racing messages can still arrive in any order, and must be buffered so that they can be received in the correct order. Buffering is normally not a problem unless buffer space is limited. Buffer overflows may occur during replay that did not originally occur. To guarantee that no overflows occur, we can reproduce the original message arrival order by passing *control* messages during replay. When a message $a \xrightarrow{M} b$ races with a message $c \xrightarrow{M} d$, a control message from a new send (added by the replay system) just after b to a new receive just before c ensures that $c \xrightarrow{M} d$ is not sent until $a \xrightarrow{M} b$ has been received. Because such messages introduce additional orderings that were not present during the original execution, they can reduce the amount of parallelism achievable during replay, and should only be used if buffer overflow is a problem.

4.3. Optimal Message Tracing

We now show that our tracing algorithm is optimal in most cases. We characterize when the traces are optimal and present example executions for which minimal and non-minimal traces are generated. Even when non-minimal traces are recorded, they are usually small.

Our tracing algorithm generates minimal traces if each message is either involved in only one race, or if messages participate in multiple races and the races are *transitive*. Transitive races often occur when receive operations specify either a single channel or any channel over which to accept a message, instead of a *subset* of channels. We prove in Appendix A (Theorem 4) that our tracing algorithm is optimal on executions for which the *RacesWith* relation is transitive (see Definition 3.2). Figure 4a illustrates such an execution. Because all three receive operations could have accepted messages from any channel, all three messages, $a \xrightarrow{M} b$, $c \xrightarrow{M} d$, and $e \xrightarrow{M} f$, race with each other. The optimal trace consists of recording any two of the three messages (such as the two shaded messages): without tracing these messages, replay cannot ensure that all three messages are delivered to the correct event. In this case, our algorithm traces the last two messages, which is an optimal trace.

Figure 4b shows an execution with non-transitive races; the first receive can accept messages only from processes 1 and 3, the second receive from processes 3 and 4, and the last receive from process 4. Because the second message could have been accepted by the first receive, the first two messages race. Similarly, because the third message could have been accepted by the second receive, the last two messages race. However, because the last message could not have been received by the first receive (it only accepts messages from processes 1 and 3), the first and last messages do *not* race. The optimal trace thus consists of recording *only* the second message: if replay ensures that the second message is delivered to the second receive operation, the other two messages will automatically arrive at the correct events (no other receives will accept them). Our tracing algorithm would trace the last two messages, which is not optimal.



(a): Transitive races
each message races with all others

(b): Non−transitive races
first and second messages race
second and third messages race
first and third messages do not race

Figure 4. (a) Transitive races, (b) non-transitive races: shaded messages show the optimal trace (tracing algorithm would trace the last two messages)

---

5. Implementation and Experimental Results

We now discuss experiences with our trace and replay strategy. We first discuss an implementation of our tracing algorithm based on appending *vector timestamps* onto user messages. These timestamps provide information about the $\xrightarrow{HB}$ relation needed to perform on-the-fly race checks. We then discuss experiments performed on a collection of message-passing programs on a 64-node Thinking Machines CM-5 and a 32-node Intel iPSC/2 hypercube. These experiments show that only 0–19% of the messages in these programs were traced, and in all but one case the optimal trace was generated. In addition, the small traces completely alleviated tracing bottlenecks that plague traditional schemes which trace every message. These results suggest that our trace and replay technique is very effective in practice, producing small traces with low execution-time overhead, providing a new technique to debug even long-running programs.

5.1. Implementation

The tracing algorithm presented in Section 4 detects races by determining the $\xrightarrow{HB}$ relation between the sender of a message and a previous receive. Our implementation of this algorithm maintains the $\xrightarrow{HB}$ relation during execution with a *vector timestamp* in each process. A vector timestamp is a vector of length *p* (the number of processes) containing event serial numbers[2]. These timestamps are maintained by appending them onto user messages and updating them after each receive operation. The tracing algorithm detects races by comparing timestamp values and event serial numbers to determine whether the previous receive happened before the sender of the current message.

In our implementation, each process maintains both a local virtual clock, *Clock*, and a vector timestamp, *Timestamp*. The local clock is used to assign serial numbers to events: events are numbered sequentially within a process beginning with the number 1, and the clock is incremented after each operation. The timestamps are maintained so that at any point during execution, the i^{th} slot of the vector timestamp for process *p* (i.e., *Timestamp*[*i*]) equals the serial number of the last event in process *i* that happened before the most recent event in process *p*. By definition, the p^{th} slot equals the current value of *p*'s local clock. To maintain these timestamps, each process appends the current value of its timestamp onto the end of each message it sends. Upon receiving a message, it updates its timestamp by computing the component-wise maximum with the timestamp appended to the incoming message.

The race check in line 4 of our tracing algorithm (Figure 4) is performed easily using the timestamps. The sender's timestamp (which is appended to the incoming message) is compared to the serial number of the previous receive to determine if the receive happened before the sender. The value of the p^{th} slot of the sender's timestamp equals the serial number of the most recent event in process *p* that happened before the sender. If the serial number of the previous receive is greater than this value, then the previous receive did not happen before the sender, and a frontier race exists.

5.2. Experimental Results

We implemented our tracing algorithm on two message-passing parallel machines: a 64-node Thinking Machines CM-5 and a 32-node Intel iPSC/2 hypercube. On each machine, two instrumented versions of the message-passing library were created. One version uses the traditional approach of tracing every message sent during execution, and the other version uses our tracing algorithm to trace only racing messages. We analyzed a collection of message-passing programs obtained from colleagues and measured two quantities. First, the percentage of messages that race was recorded. This percentage shows the trace size reduction obtained by our race-based tracing strategy. Second, the increase in execution time of both the traditional approach of tracing every message and our approach of tracing only racing messages was measured. These overheads show whether the cost of performing race checks outweighs the time savings obtained by not tracing non-racing messages. We found that often only 0–2% of the total messages were traced, and in cases where the execution-time overhead of tracing every message is high, race-based tracing is an order of magnitude faster.

Table 1 shows the results of our experiments on six programs. *det* computes the determinant of a matrix, and was run on a randomly generated 100×100 matrix. *line* computes the intersections of a collection of line segments, and was run on 1000 randomly generated segments. *mesh* computes finite differences over a grid to solve a differential equation, and was run on a 300×300 grid. *mult* multiplies two matrices, and was run on two randomly generated 100×100 matrices. *sys* uses Gaussian elimination to solve a linear system of equations, and was run on a system of 300 randomly generated (linearly independent) equations. *tycho* is a cache simulator, and was run on a 10 MByte address trace.

Our first experimental result pertains to trace sizes: In two programs (*det* and *sys*), only 1–2% of the messages were traced — a two order of magnitude reduction over tracing every message. In two programs (*line*, *mult*), *none* of the messages raced, and no traces at all were generated. Executions of these programs (on the given input) are guaranteed to be reproduced automatically; nothing special need be done during replay. In two programs (*mesh*, *tycho*), 14–19% of the messages were traced. These cases represent programs that were designed to be highly internally nondeterministic (although their final results are deterministic); they use some form of a first-come first-served worker paradigm. Even in such cases, the number of racing messages was rather low. In five of the six programs the optimal trace was generated (because the races were transitive, as discussed in Section 4.3). In the other program (*tycho*), the trace size was within 46% (or less) of optimal†. These results suggest that our tracing strategy is effective, tracing as few as 0–2% of the messages, and no more than 19% of the messages even in programs that are highly nondeterministic. In addition, our tracing algorithm generated optimal traces for most of the test programs, and even when non-optimal traces were generated, they were small.

Program	**Messages Sent**	**Messages Traced**	**% of Opt. Trace**	**Overhead all msgs**	**Overhead racing msgs**
det‡	4713	63 (1%)	optimal	568%	8%
line‡	31	0 (0%)	optimal	0.3%	8%
mesh‡	10210	1392 (14%)	optimal	28%	14%
mult‡	1120	0 (0%)	optimal	15%	0.1%
sys‡	9424	332 (2%)	optimal	561%	14%
tycho†	1791	412 (19%)	within 46%	–	–

† 64-node Thinking Machines CM-5
‡ 32-node Intel iPSC/2

Table 1. Results of message tracing

Our second result pertains to the execution-time overhead incurred by our tracing strategy. To assess this overhead, we analyzed three versions of each program: the original (uninstrumented) program, an instrumented version that traces every message, and an instrumented version that traces only racing messages. Each version of every program was executed 10 times and the average execution times were computed. The last two columns of Table 1 show the execution-time overhead incurred by tracing all messages and tracing only racing messages‡.

Two programs (*det* and *sys*) suffered substantial slowdown (almost 600%) when using the traditional approach of tracing every message. These programs exhibited a high frequency of message operations. Tracing this high traffic introduced a bottleneck, since many trace records needed to be written in a short time. In contrast, our strategy of tracing only the racing messages reduced the tracing requirements to the point where the bottleneck was completely alleviated (resulting in a slowdown of only 8–14%). In the programs that had no racing messages (*line*, *mult*), the overhead of race-based tracing indi-

† We derived this quantity by computing a lower bound on the optimal trace size. As shown in the proof of Theorem 3 (in Appendix A), computing an optimal trace is equivalent to computing a minimal vertex cover of a graph. To determine how the recorded trace size compares to the optimal trace, we used graph matching to estimate the optimal size to within a factor of two. For *tycho*, the recorded trace size was *no more* than 46% larger than optimal. Determining whether it was actually optimal would require computing a minimal vertex cover, which is an intractable problem in general.

‡ Because our CM-5 had not yet been equipped with I/O processors, overhead measurements for this machine were not reported. Our implementation performed trace I/O by sending messages over the diagnostic network. The unusually high cost of this I/O makes the overhead of tracing all messages several orders of magnitude higher than race-based tracing, making comparisons unrealistic.

cates the inherent cost of performing the on-the-fly race checks. This overhead shows that the cost of maintaining the vector timestamps and checking for races is low (0.1-8%). Because *line* had low message-passing traffic, tracing every message was cheaper than maintaining the timestamps. In *mult*, longer messages were passed between nodes, making the incremental cost of appending timestamps low. In both programs, tracing every message did not introduce a bottleneck, and both tracing strategies had reasonable overheads.

6. Conclusions

In this paper we presented a trace and replay strategy for message-passing parallel programs that is often optimal and always effective. The small traces and low overhead produced by this strategy allow even long-running programs to be replayed that previously could not have been replayed in practice. This work provides a new foundation on which efficient parallel-program debugging techniques can be built. We achieve these benefits by making tracing decisions at run-time, instead of tracing every message (as earlier work proposes). Race checks are performed after each receive operation to locate and trace exactly those messages that introduce nondeterminacy. We prove that our tracing strategy is optimal for many programs which exhibit simple message patterns. Even when non-optimal traces are generated, experiments show that the traces are kept small, and are one to two orders of magnitude smaller than traces of every message.

Future work includes a more precise characterization of when our algorithm is optimal (e.g., transitive races are sufficient for an optimal trace but not necessary). Better tracing strategies may be possible. By buffering more information about recent messages, more informed tracing decisions might trace fewer races. By employing optimizations for maintaining timestamps, the overhead of passing timestamps and performing race checks might also be reduced.

References

[1] Curtis, R. and L. Wittie, ‘‘BugNet: A Debugging System for Parallel Programming Environments,’’ *Proc. of the 3rd Intl. Conf. on Dist. Computing Systems*, pp. 394-399 (1982).

[2] Fidge, C. J., ‘‘Partial Orders for Parallel Debugging,’’ *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183-194 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[3] Lamport, Leslie, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *CACM* **21**(7) pp. 558-565 (July 1978).

[4] LeBlanc, Thomas J. and John M. Mellor-Crummey, ‘‘Debugging Parallel Programs with Instant Replay,’’ *IEEE Trans. on Computers* **C-36**(4) pp. 471-482 (April 1987).

Appendix A. Proofs of Theorems

Theorem 1 (Tracing Theorem).

The tracing algorithm (Figure 3) traces at least one message in each frontier race.

Proof. We prove below that a message is traced by the algorithm when some predicate is true (Lemma 1), and then prove that this predicate is true when a frontier race exists (Lemma 2). At least one message in each frontier race is thus traced. ■

Lemma 1.

If two messages, $Send \xrightarrow{M} Recv$ and $PSend \xrightarrow{M} PRecv$, exist such that $PRecv \overset{HB}{\not\longrightarrow} Send \wedge PRecv \xrightarrow{XO} Recv \wedge SEND(Send) \cap RECEIVE(PRecv) \neq \emptyset$, then the tracing algorithm traces $Send \xrightarrow{M} Recv$.

Proof. To establish a contradiction, assume that the above conditions hold but the algorithm does not trace $Send \xrightarrow{M} Recv$. This message is not traced only if the algorithm finds that the previous receive, *PrevRecv* (located in line 2), happened before the sender: $PrevRecv \xrightarrow{HB} Send$. We must also have $PRecv \xrightarrow{XO} PrevRecv$, otherwise the algorithm would find *PRecv* as the previous receive. These orderings imply that $PRecv \xrightarrow{HB} Send$, which contradicts the assumption that $PRecv \overset{HB}{\not\longrightarrow} Send$. ■

Lemma 2.

If $PSend \xrightarrow{M} PRecv$ *RacesWith* $Send \xrightarrow{M} Recv$, then $PRecv \overset{HB}{\not\longrightarrow} Send \wedge PRecv \xrightarrow{XO} Recv \wedge SEND(Send) \cap RECEIVE(PRecv) \neq \emptyset$.

Proof. Because $PSend \xrightarrow{M} PRecv$ *RacesWith* $Send \xrightarrow{M} Recv$, there exists a $P' = \langle E', \xrightarrow{HB'} \rangle \in F_{SAME}$ such that $Send \xrightarrow{M'} PRecv$ (by Definition 3.2). We consider each term in the conjunct $PRecv \overset{HB}{\not\longrightarrow} Send \wedge PRecv \xrightarrow{XO} Recv \wedge SEND(Send) \cap RECEIVE(PRecv) \neq \emptyset$.

(1) To establish a contradiction, assume that $PRecv \xrightarrow{HB} Send$. In addition, by the definition of F_{SAME} (part (3) of Definition 3.1), $Send \xrightarrow{M'} PRecv$ implies that $\forall x \in E,\ x \xrightarrow{HB} Send \Leftrightarrow x \xrightarrow{HB'} Send$. Thus, if $PRecv \xrightarrow{HB} Send$, we must have $PRecv \xrightarrow{HB'} Send$, which contradicts the assumption that $Send \xrightarrow{M'} PRecv$.

(2) To establish a contradiction, assume that $Recv \xrightarrow{XO} PRecv$. Then, P' cannot belong to F_{SAME}, since by its definition $Recv \xrightarrow{XO} PRecv$ implies that *Recv* is before the frontier and thus $Send \xrightarrow{M'} Recv$ (since $Send \xrightarrow{M} Recv$), which contradicts the assumption that $Send \xrightarrow{M'} PRecv$.

(3) Since $Send \xrightarrow{M'} PRecv$, we clearly have $SEND(Send) \cap RECEIVE(PRecv) \neq \emptyset$. ■

Theorem 2 (Replay Theorem).

By forcing the traced messages to be delivered to the same events during replay as during the original execution, all messages will be delivered to the correct events.

Proof. We prove below that any execution is deterministic if it is free of frontier races. Because at least one message of each frontier race is traced (Theorem 1), enforcing the traced messages during replay ensures that replay exhibits no frontier races. Replay will thus be deterministic, causing all messages to be delivered to the same events as during the original execution.

To establish a contradiction, assume that some program execution $P = \langle E, \xrightarrow{HB} \rangle$ is nondeterministic but free of frontier races. Since P is nondeterministic, another execution of the program on the same input could produce a different execution, $P' = \langle E', \xrightarrow{HB'} \rangle$. P and P' exhibit the same events and message deliveries up to some point after which they differ. Let r be a receive event where they first differ. That is, $x \xrightarrow{HB} y \Leftrightarrow x \xrightarrow{HB'} y$ for all events x,y where $x \xrightarrow{HB} r$ and $y \xrightarrow{HB} r$. Also let s_1 and s_2 be operations that send messages to r in P and P': $s_1 \xrightarrow{M} r$ and $s_2 \xrightarrow{M'} r$. The messages sent by s_1 and s_2 in P must race because P' meets the conditions in Definition 3.1 and thus belongs to F_{SAME}. The events x and y are all the events before the frontier, and s_1, s_2, and r are after the frontier. But P containing a frontier race contradictions the assumption. Thus, P' cannot be different than P, implying that P is deterministic. ■

Theorem 3 (Tracing Complexity Theorem).

Given a program execution, $P = \langle E, \xrightarrow{HB} \rangle$, determining whether replay can be implemented by tracing k or fewer messages is an NP-hard problem.

Proof. We use a reduction from the vertex cover problem, known to be NP-complete: given an undirected graph, $G = (V,E)$, does G have a vertex cover with k or fewer vertices? A vertex cover is a subset V' of the vertices such that every edge is connected to some vertex in V'. Given a graph, G, we reduce the problem of determining whether it has a vertex cover with k or fewer vertices to the problem of determining whether a program execution, P, can be replayed from a trace of k or fewer messages.

From the graph G we construct P as follows. P contains two processes between which a message is sent for each of the n vertices in G. Process 1 in P contains n send operations, and process 2 contains n receive operations. The i^{th} send operation sends a null message over logical channel i, and the i^{th} receive operation specifies that it will receive over logical channel i. Additional channels are specified by the receive operations so that two messages race iff an edge connects their corresponding nodes in G. For an edge from vertex i to vertex j, the i^{th} receive operation also specifies that it will receive over logical channel j. Because messages sent from process 1 may be delivered out of order, the i^{th} and j^{th} messages in P race iff an edge exists from vertex i to vertex j in G.

G has a vertex cover with k or fewer vertices iff P can be replayed from a trace of k or fewer messages. Assume that G has a vertex cover V' with k vertices. Each vertex in V' corresponds to one of the messages sent by P. P can be replayed from a trace of exactly these messages. Since two messages race iff an edge connects their corresponding vertices, a vertex cover ensures that at least one message in each race is traced. By the Replay Theorem (Theorem 2), a trace of these messages suffices for replay. Conversely, assume that P can be replayed from a trace T of k messages. T must contain at least one message in each frontier race, or else replay will not be frontier-race-free. Since two messages race in P iff an edge connects the corresponding vertices in G, the vertices corresponding to the messages in T are a vertex cover. ■

Theorem 4 (Optimality Theorem).

For any program execution, $P = \langle E, \xrightarrow{HB} \rangle$, for which the *RacesWith* relation is transitive, the tracing algorithm (Figure 4) traces a minimal number of messages required to implement replay.

Proof. As in Theorem 3, we can view the tracing problem as equivalent to computing a vertex cover of a graph. The program execution P defines a graph G: the messages in P define its vertices, and when two messages race an edge is drawn between the corresponding vertices. As discussed in Theorem 3, any trace sufficient for replay must cover the vertices of G. When the *RacesWith* relation is transitive, G becomes a forest of completely connected graphs. In this case, a minimal vertex cover is easily computed. For each completely connected component of n vertices, a minimal vertex cover consists of any $n-1$ vertices. When races are transitive, our tracing algorithm traces all but one of the mutually racing messages (the first racing message is not traced), which corresponds to such a minimal vertex cover. Thus, the trace is optimal. ■