Efficient Race Detection for Message-Passing Programs with Nonblocking Sends and Receives

Robert Cypher

Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
cypher@cs.jhu.edu

Eric Leu

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
leu@almaden.ibm.com

Abstract

This paper presents an algorithm for performing on-the-fly race detection for parallel message-passing programs. The algorithm reads a trace of the communication events in a message-passing parallel program and either finds a specific race condition or reports that the traced program is race-free. It supports a rich message-passing model, including blocking and nonblocking sends and receives, synchronous and asynchronous sends, receive selectivity by source and/or tag value, and arbitrary amounts of system buffering of messages. It runs in polynomial time and is very efficient for most types of executions. A key feature of the race detection algorithm is its use of several new types of logical clocks for determining ordering relations. It is likely that these logical clocks will also be useful in other settings.

1 Introduction

Many commercial parallel computers support a message-passing model in which processes communicate solely by issuing matching send and receive commands. If a single receive command can be matched to several different send commands (or vice-versa), the program is said to have a *race condition*. The detection of race conditions is critical for debugging parallel programs for two reasons. First, although some parallel programs are designed to have race conditions, many (including all deterministic programs) are designed to be free of race conditions. Thus the existence of a race condition can be proof of an error in the program. Second, race conditions complicate debugging because their nondeterministic nature can prevent incorrect program behavior from being repeated.

In addition to the standard blocking sends and receives, many message-passing environments now provide nonblocking send and/or receive commands [6, 10, 13, 16, 23, 27]. The nonblocking commands support the overlapping of communication and computation, and they can be used to avoid deadlock [5]. However, because nonblocking commands allow a single process to have many outstanding sends and receives at a single time, their use greatly increases the potential for race conditions. As a result, the detection of race conditions in parallel programs that use nonblocking sends and receives is particularly important.

Other researchers have studied race conditions in parallel programs. Damodaran-Kamal and Francioni created a tool for detecting race conditions in message-passing programs that use PVM [8]. However, they assume that only blocking sends and receives are used. Researchers have also considered race conditions in the context of execution replay [4, 15, 19, 20, 24, 26], where the goal is to record enough information to reproduce an execution. In general, execution replay aids debugging by allowing bugs to be reproduced, but it does not help detect races in programs which should be race-free. An exception is the execution replay work by Netzer [24] and by Netzer and Miller [26], as they detect races in order to reduce the amount of information that is recorded. However, the work by Netzer [24] is for shared memory programs, while the work by Netzer and Miller [26] applies only to message-passing programs with blocking sends and receives. Finally, many researchers have studied race conditions in parallel programs that use shared memory [1, 2, 9, 11, 14, 17, 22, 24, 25].

In this paper we describe an algorithm for detecting race conditions in parallel programs. It is a sequential algorithm that reads a trace of the communication events in a message-passing parallel program and determines if the execution contains a race condition. The test is exact, so if the execution is race-free, the algorithm will always report that there are no race conditions, and if the execution contains a race condition, the algorithm will always find and report one such race. We assume a rich message-passing model with both blocking and nonblocking sends and receives, synchronous and asynchronous sends, receive selectivity by source and/or tag value, and arbitrary amounts of system buffering of messages. Our algorithm can detect races in arbitrary executions, including ones in which some or all of the processes deadlock (which is important for debugging). It runs in polynomial time and is extremely efficient for most types of executions. In addition, it operates on-the-fly, thus greatly reducing its storage requirements (as only a

1063-6374/95 $04.00 © 1995 IEEE

small set of the most recently traced information is maintained at any time). An important feature of our race detection algorithm is its use of several new types of logical clocks for determining ordering relations. These logical clocks, which are fundamentally different from those previously proposed [12, 14, 18, 21, 26], are the first to capture the ordering information present when nonblocking sends and receives are used. As a result, it is likely that they will also be useful in other settings.

The remainder of this paper is organized as follows. Section 2 discusses the message-passing model, the parameters that are used for analyzing the algorithm, race conditions, and known results. The new types of logical clocks are presented in Section 3. Section 4 describes the race detection algorithm. Conclusions and directions for future research are given in Section 5. Some proofs are omitted due to space limitations.

2 Preliminaries

2.1 Send and Receive Primitives

We will consider both blocking and nonblocking sends and receives and both synchronous and asynchronous sends. A *blocking-asynchronous-send* does not return until the message has been copied out of the sender's buffer, while a *blocking-synchronous-send* does not return until the message has been delivered to the receiver. The nonblocking versions of these send commands always return immediately without copying the message out of the sender's buffer. The sender cannot overwrite that buffer until he issues a corresponding *wait-for-send* command. Nonblocking receives are analogous in that they return immediately, even if no message has arrived. The receiver must issue a corresponding *wait-for-receive* command before reading the received message.

A receive command selects messages by specifying either a single source or a wildcard value that matches any source, and either a single tag value or a wildcard value that matches any tag[1]. In this paper we will assume *weakly-ordered* communication, which was defined by Cypher and Leu [6] and has been adopted in the Message-Passing Interface (MPI) standard [23]. Informally, given any receive command, weakly-ordered communication requires that only the first send that is still unmatched and is compatible with the given receive can be matched to that receive. Similarly, given any send command, weakly-ordered communication requires that only the first receive that is still unmatched and is compatible with the given send can be matched to that send.

2.2 Formal Model

All of the results in this paper are based on a formal model of message-passing semantics [6]. We will now give a brief overview of that model. We consider the execution of a message-passing program that consists of p static processes. A *program execution* is represented by a pair $\langle E, M \rangle$, where

- $E = E_1, E_2, \ldots, E_p$ and each E_i, $1 \leq i \leq p$, is the finite sequence of events executed by process i, and
- M is a set of ordered pairs of the form (a, b), where a and b are both events in E, which specifies the matching of send events to receive events.

Each event $e \in E$ is one of the following:

PS: A PS (Post-Send) event is a nonblocking send of a message,

WS: A WS (Wait-for-Send-to-complete) event is a synchronous wait for a specific earlier send event,

WB: A WB (Wait-for-Buffer) event is an asynchronous wait for a specific earlier send event,

PR: A PR (Post-Receive) event is a nonblocking receive of a message,

WR: A WR (Wait-for-Receive-to-complete) event is a wait for a specified earlier receive event,

INTERNAL: An INTERNAL event consists of a sequence of local calculations that do not perform any message-passing, and

FINAL: A FINAL event indicates that the process executing the event terminated successfully (without deadlocking).

The events in the formal model map directly to nonblocking sends and receives and their associated wait commands. However, blocking sends and receives can be represented by a pair of events. For example, a *blocking-asynchronous-send* maps to a PS immediately followed by a WB. The FINAL event is a formality which allows the specification of both deadlocking and nondeadlocking processes. A match $m = (a, b) \in M$ is a pairing of a PS a with a PR b indicating that the message sent by a is received by b. The formal model makes no assumptions about the availability of system buffers, so a WB may return immediately or it may block until its corresponding PS has been matched to a PR.

The following notation will be used throughout this paper. Given any event e:

- $kind(e)$: is PS, WS, WB, PR, WR, INTERNAL, or FINAL,
- $proc(e)$: denotes the process that executed e,
- $corr(e)$: denotes the corresponding event executed at process $proc(e)$ (e.g., if $kind(e)$ = PR, $corr(e)$ denotes e's corresponding WR, and if $kind(e)$ = WB, $corr(e)$ denotes e's corresponding PS), and

Given any PS e, $dest(e)$ will denote e's destination process and $tag(e)$ will denote e's tag. Given any PR e, $source(e)$ will denote e's source process parameter and $tag(e)$ will denote e's tag parameter (either or both of which may have the value "wildcard").

[1] A tag is an integer that is associated with each message when it is sent.

A PS e is *matched* if there exists a PR f such that $(e, f) \in M$, and it is *unmatched* otherwise. Similarly, a PR e is *matched* if there exists a PS f such that $(f, e) \in M$, and it is *unmatched* otherwise.

Given any events a and b, a *immediately precedes* b iff b immediately follows a in the sequence $E_{proc(a)}$, and a *precedes* b iff b appears later in the sequence $E_{proc(a)}$ than does a. Given any PS s and any PR b, a and b are *compatible* iff s's destination is $proc(r)$, r's source is $proc(s)$ or a wildcard value, and r's tag is s's tag or a wildcard value.

Given any relation R, the transitive closure of R will be denoted R^+ and the reflexive and transitive closure of R will be denoted R^*.

2.3 Parameters

The following parameters will be used to analyze algorithms throughout this paper.

- N_e: number of events in entire trace
- N_p: number of processes
- N_s: maximum number of events local to any one process
- N_t: maximum number of distinct tags values used per process
- N_x: maximum number of overlapping posted sends or receives (see below)

The parameter N_x captures how many PS or PR events can be active simultaneously at a single process. More precisely, given a PS s, let $count(s)$ denote the number of other PS events s' such that s precedes s' and s' precedes $corr(s)$ (if $corr(s)$ exists). Given any PR r, define $count(r)$ analogously. The parameter N_x is the maximum, over all PS and PR events x, of $count(x)$. Note that typically $N_t \ll N_s$ and $N_x \ll N_s$.

2.4 Race Conditions

Intuitively, a program execution $\langle E, M \rangle$ has a race condition if there exists a PS s and a PR r that were not matched to one another, but could have been matched to one another. This notion is defined formally [7] by considering every possible PS s and PR r which were not matched to one another, and considering every possible intermediate state of the program execution. If at any possible intermediate state s can be matched to r without violating the model's semantics, there is a race condition.

Unfortunately, there could be an exponential number of possible intermediate states, so the formal definition does not immediately provide a polynomial time race detection algorithm. Fortunately, a characterization of race conditions has been created that does lead to a polynomial time algorithm [7]. This characterization will be given in Theorem 2.2 below.

2.5 Known Results

The following theorem captures a property of weakly-ordered communication [7].

Theorem 2.1 *Given any matches (s_1, r_1) and (s_2, r_2), if s_1 precedes s_2 and r_2 precedes r_1, then s_1 and r_2 are not compatible.*

The following ordering relations will be needed for the efficient characterization of race conditions.

Given matches (s_1, r_1) and (s_2, r_2), $R_m((s_1, r_1), (s_2, r_2))$ iff:

- s_1 and r_2 are compatible and s_1 precedes s_2, or
- s_2 and r_1 are compatible and r_1 precedes r_2.

The "happened-immediately-before" relation for message-passing systems with weakly-ordered communication, R_h, is defined as follows:

1. Given events x and y, $R_h(x, y)$ iff:
   - x precedes y.
2. Given an event x and a match (s, r), $R_h(x, (s, r))$ iff:
   - $x = s$ or $x = r$.
3. Given a match (s, r) and an event x, $R_h((s, r), x)$ iff:
   - $corr(s)$ immediately precedes x and $corr(s)$ is a WS, or
   - $corr(r)$ immediately precedes x.
4. Given matches (s_1, r_1) and (s_2, r_2), $R_h((s_1, r_1), (s_2, r_2))$ iff:
   - $R_m((s_1, r_1), (s_2, r_2))$.

Intuitively, these rules capture the ordering relationships between the starting times of events and matches. Rule 1 captures the fact that events are executed sequentially per process. Rule 2 states that a match between PS s and PR r cannot occur until events s and r have begun execution. Rule 3 states that the event following a WS (respectively, WR) cannot begin execution until the PS (PR) event for which the WS (WR) is waiting has been matched. Rule 4 captures the message-ordering properties of weakly-ordered communication. Finally, relation R_h^+ is the "happened-before" relation for message-passing with weakly-ordered communication, and is analogous to Lamport's "happened-before" relation [18]. However, unlike Lamport's "happened-before" relation, R_h^+ is a partial order of the events *and* the matches, rather than simply events. Matches are included in this relation because matches depend on earlier events (by Rule 2), matches depend on earlier matches (by Rule 4), and events depend on earlier matches (by Rule 3). The fact that matches must be included in the "happened-before" relation will have a significant impact on the detection of race conditions.

Definition: Given any match (s_1, r_1) and any PS s_2, (s_1, r_1) and s_2 *conflict* iff

- s_2 and r_1 are compatible,

- s_1 and s_2 were executed by different processes,
- $\neg R_h^+((s_1, r_1), s_2)$, and
- s_2 is not matched to a PR r_2 where r_2 precedes r_1.

Theorem 2.2 gives a precise characterization of those programs that have race conditions [7].

Theorem 2.2 *A program execution $\langle E, M \rangle$ has a race condition iff there exists a match (s_1, r_1) and a PS s_2 which conflict.*

Given Theorem 2.2 it is relatively simple to create an $O(N_e^2 N_s^2)$ time off-line race detection algorithm. The main bottleneck in this algorithm is testing if $R_h^+((s_1, r_1), s_2)$. This bottleneck can be overcome with the use of logical clocks, as will be described in the next section.

3 Logical Clocks

A logical clock is a value that is associated with each element of a partially ordered set in order to speed the comparison of items in the set [12, 14, 18, 21, 26]. In this section we will see how logical clocks can be created for events and matches in the context of on-the-fly race detection.

3.1 Type A Clocks

The first type of logical clock we will consider is a natural extension of the vector clocks that were created for message-passing systems with blocking sends and receives [12, 21]. These clocks, which we call Type A clocks, are defined below.

Given any vectors u and v, let $sup(u, v)$ denote the component-wise maximum of u and v. Given any vector u and index i, let $inc(u, i)$ denote the vector v where $v[i] = u[i] + 1$ and $v[j] = u[j]$ for $j \neq i$.

Each event or match x is assigned a clock A^x which is a vector of N_p integers. Type A clocks are calculated as follows:

- If y is the first event in some process i:
  - $A^y[i] := 1$;
  - $A^y[j] := 0$ for all $j \neq i$.
- If x and y are events in some process i where x immediately precedes y:
  - $A^y := inc(A^x, i)$;
  - if there exists a match (s, r) such that $R_h((s, r), y)$:
    $A^y := sup(A^y, A^{(s,r)})$.
- If (s, r) is a match:
  - $A^{(s,r)} := sup(A^s, A^r)$;
  - for all matches (s', r') such that $R_h((s', r'), (s, r))$:
    $A^{(s,r)} := sup(A^{(s,r)}, A^{(s',r')})$.

Intuitively, given any event or match x, $A^x[i]$ gives the number of events in process i that must have begun execution when x began execution. The following theorem shows that Type A clocks can be used to determine certain R_h^+ relations. The proof is given in the appendix.

Theorem 3.1 *Given any event a and any event or match b where $a \neq b$,*

$$R_h^+(a, b) \Leftrightarrow A^a[proc(a)] \leq A^b[proc(a)].$$

Note that Theorem 3.1 provides an efficient means for determining if $R_h^+(a, b)$ where a is an *event* and b is a *match*. However, it does not allows us to efficiently determine if $R_h^+(a, b)$ where a is a *match* and b is an *event*, which is what is required for detecting race conditions. This is because A^x provides information about the events that must happen before x, but not about the matches that must happen before x. One approach to solving this problem would be to add to A^x information about matches that must have happened before x. However, this cannot be done efficiently because the matches are only partially ordered with respect to one another, even if we restrict our attention to the matches that were either sent from or received at a single process. In contrast, the events that are executed at a single process are totally ordered with respect to one another, which is why the clock A^x can encode the set of events at process i that must have happened before x in the single integer $A^x[i]$.

3.2 Type B Clocks

We will now introduce logical clocks that can be used to test if $R_h^+(a, b)$ where a is a match and b is an event. These clocks, which we call Type B clocks, are similar to Type A clocks, except they record information about the number of events in each process that must have begun execution *after* a given match or event began execution. Type B clocks are calculated as follows:

- If x is the last event in some process i:
  - $B^x[i] := 1$;
  - $B^x[j] := 0$ for all $j \neq i$.
- If x and y are events in some process i where x immediately precedes y:
  - $B^x := inc(B^y, i)$;
  - if there exists a match (s, r) such that $R_h(x, (s, r))$:
    $B^x := sup(B^x, B^{(s,r)})$.
- If (s, r) is a match:
  - $B^{(s,r)}[j] := 0$ for all j;
  - for all events y such that $R_h((s, r), y)$:
    $B^{(s,r)} := sup(B^{(s,r)}, B^y)$.

– for all matches (s', r') such that $R_h((s, r), (s', r'))$:
$B^{(s,r)} := sup(B^{(s,r)}, B^{(s',r')})$.

Theorem 3.2 *Given any event or match a and any event b where $a \neq b$,*

$$R_h^+(a, b) \Leftrightarrow B^a[proc(b)] \geq B^b[proc(b)].$$

As a result of Theorem 3.2, Type B clocks can be used to create an efficient off-line race detection algorithm. However, the Type B clock of an event or match x depends on the clocks of events and matches that happened after x, so Type B clocks are not well-suited to on-the-fly race detection. To be more precise, it is possible to use Type B clocks for on-the-fly race detection, but the algorithm has prohibitive storage requirements because the clocks cannot be discarded, due to the fact that events and matches that occur much later could cause them to be updated. What is needed for on-the-fly race checking is a type of clock that depends only on the clocks of earlier events and actions, plus the clocks of at most a small number of later events and actions.

3.3 Type C Clocks

We will now formally define Type C clocks. Assume that each event or match x has a Type A clock A^x as defined above. Given any match (s, r), $C^{(s,r)}$ is calculated as follows:

- $C^{(s,r)}[j] := \infty$ for all j;
- for all matches (s', r') such that $R_m^*((s, r), (s', r'))$ and for all events y such that $R_h((s', r'), y)$:
$C^{(s,r)}[proc(y)] := \min(C^{(s,r)}[proc(y)], A^y[proc(y)])$.

The following theorem is proven in the appendix.

Theorem 3.3 *Given any match (s, r) and any event x, $R_h^+((s, r), x) \Leftrightarrow \exists j$ such that $C^{(s,r)}[j] \leq A^x[j]$.*

As a result of Theorem 3.3, Type A and C clocks can be used for race detection. Note that Type C clocks are fundamentally different from all previously defined logical clocks [12, 14, 18, 21, 26], as they must be used in conjunction with another type of clock in order to test the "happened-before" relation.

3.4 Type D Clocks

Type C clocks still have the disadvantage that $C^{(s,r)}$ can depend on events and matches that happen much later than (s, r). However, by making a small change to these clocks, we will be able to remove this problem. The resulting clocks, called Type D clocks, are defined formally below. Assume that each event or match x has a Type A clock A^x as defined above. Given any match (s, r), $D^{(s,r)}$ is calculated as follows:

- $D^{(s,r)}[j] := \infty$ for all j;
- for all matches (s', r') such that $R_m^*((s, r), (s', r'))$ and $corr(r)$ does not precede r', and for all events y such that $R_h((s', r'), y)$:
$D^{(s,r)}[proc(y)] := \min(D^{(s,r)}[proc(y)], A^y[proc(y)])$.

The following theorem shows how Type A and D clocks can be used for race detection.

Theorem 3.4 *Given any match (s, r) and any event x, $R_h^+((s, r), x) \Leftrightarrow \exists j$ such that $D^{(s,r)}[j] \leq A^x[j]$.*

The following theorem characterizes the types of matches that can be related by the R_m relation.

Theorem 3.5 *Given any matches (s_1, r_1) and (s_2, r_2), if $R_m((s_1, r_1), (s_2, r_2))$ then r_1 precedes r_2.*

As a result of Theorem 3.5, it follows that for any match (s, r), there are at most N_x matches (s', r') such that $R_m^+((s, r), (s', r'))$ and $corr(r)$ does not precede r'. Therefore, given any match (s, r), $D^{(s,r)}$ depends only on the clocks of at most $O(N_x)$ events.

4 On-the-fly Race Detection

We will now show how Type A and D clocks can be used to create an efficient on-the-fly race detection algorithm. Throughout this section we will assume that the events and matches are read by the race detection program in a *consistent* order, that is in some total order that is consistent with the R_h^+ relation.

4.1 Data Structures

Our race detection algorithm maintains N_p linked lists of events, denoted $L_1, L_2, \ldots, L_{N_p}$, where L_i contains the events executed by process i stored in the order in which they were executed. Each event x has a pointer to its corresponding event, $corr(x)$. The algorithm also maintains the matches that have occurred, and it maintains pointers between each match (s, r) and the events s and r. A Type A clock is maintained for each event and match, and a Type D clock is maintained for each match.

4.2 Calculating the Logical Clocks

Whenever a new event or match x is read by the race detection algorithm, A^x is calculated. It is possible to calculate A^x immediately upon reading x because A^x depends only on those clocks A^y such that $R_h(y, x)$, so A^y has already been calculated. If x is an event, A^x is a function of events and matches that can be located in $O(1)$ time using the data structures described above. If $x = (s, r)$ is a match, it is necessary to locate each match (s', r') such that $R_m((s', r'), (s, r))$. It follows from Theorem 3.5 that r' precedes r. In fact, the following theorem shows that only the N_x PRs preceding r are involved in matches that need to be checked in order to calculate A^x.

Theorem 4.1 *Given any matches (s, r) and (s', r') where $corr(r')$ precedes r, $A^r[j] \geq A^{(s',r')}[j]$ for each process j.*

Proof: Let j be an arbitrary process and let z be the event such that $corr(r')$ immediately precedes z. Note that $R_h((s', r'), z)$ and z precedes r. Therefore, it follows from the definition of Type A clocks that $A^{(s',r')}[j] \leq A^z[j] \leq A^r[j]$. □

In addition, we must calculate the Type D clocks for the matches. When a match (s, r) is read, each entry in its clock $D^{(s,r)}$ is initialized to infinity. The clock $D^{(s,r)}$ must then be updated with the Type A clocks of certain events. However, those events come after (s, r), so they are not known when (s, r) is read. Instead, when each event y is read and A^y has been calculated, if there is a match (s', r') such that $R_h((s', r'), y)$ we search for each match (s, r) such that $R_m^*((s, r), (s', r'))$ and $corr(r)$ does not precede r' and set $D^{(s,r)}[proc(y)] := \min(D^{(s,r)}[proc(y)], A^y[proc(y)])$. It follows from Theorem 3.5 that $proc(r) = proc(r')$, so only those matches received at a single process have to be tested to see if their Type D clocks should be updated with A^y. Furthermore, it follows from the definition of Type D clocks that only $O(N_x)$ such matches have to be tested.

4.3 Detecting Races

In order to detect race conditions, for each match (s_1, r_1) and PS s_2, we must check if (s_1, r_1) and s_2 conflict. Given a PS s_2 we will check the $O(N_s)$ matches (s_1, r_1) such that $proc(r_1) = dest(s_2)$ to see if (s_1, r_1) and s_2 conflict. We cannot do this when we read PS s_2, because the conflicting match (s_1, r_1) might not yet have been read. However, it follows from the following theorem that if s_2 is matched to a PR r_2, a conflicting match (s_1, r_1) must have been read before (s_2, r_2) is read.

Theorem 4.2 *Given any matches (s_1, r_1) and (s_2, r_2), if (s_1, r_1) and s_2 conflict, then $R_h^+((s_1, r_1), (s_2, r_2))$.*

Proof: It follows from the definition of conflict that s_2 and r_1 are compatible and r_1 precedes r_2. Therefore, $R_m((s_1, r_1), (s_2, r_2))$, which implies $R_h^+((s_1, r_1), (s_2, r_2))$. □

Thus whenever a match (s_2, r_2) is read, the $O(N_s)$ matches (s_1, r_1) such that r_1 precedes r_2 are tested to see if (s_1, r_1) and s_2 conflict. This will detect if (s_1, r_1) and s_2 conflict for any matched PS s_2. Possible conflicts involving unmatched PS events are tested after all of the inputs have been read.

It should be noted that the above tests use the Type D clocks of earlier matches in order to detect race conditions, despite the fact that the Type D clocks are initially set to infinity and are modified as later events are read. Fortunately, it can be shown that because the events and matches are read in a consistent order, the above tests will always be correct (even if the Type D clocks have not been completely calculated).

4.4 An Improvement

The algorithm described above can be improved due to the following observation. It has been shown that only the $O(N_x)$ most recent events per process, and their associated matches, need to be maintained for the calculation of Type A and D clocks. Thus, each list L_i will only hold the most recent $O(N_x)$ events, and events no longer on these lists will be deleted. In addition, matches involving events that have been deleted can also be deleted, unless they are still needed for race detection. It is immediate from the definition of a conflict that if (s_1, r_1) and s_2 conflict, then $source(r_1)$ must be a wildcard, so the match (s_1, r_1) is not needed for race detection if $source(r_1)$ is not a wildcard. In fact, the following theorems show that even among the matches with wildcard sources, only a few cannot be deleted.

Theorem 4.3 *Given any matches (s_1, r_1), (s_1', r_1'), and (s_2, r_2), where $source(r_1) = source(r_1') =$ wildcard, $tag(r_1) = tag(r_1')$, and r_1 precedes r_1' which precedes r_2, if (s_1, r_1) and s_2 conflict then either (s_1', r_1') and s_2 conflict or (s_1, r_1) and s_1' conflict.*

Theorem 4.4 *Given any matches (s_1, r_1) and (s_1', r_1') where $source(r_1) = source(r_1') =$ wildcard, $tag(r_1) = tag(r_1')$, and r_1 precedes r_1', and given any unmatched PS s_2, if (s_1, r_1) and s_2 conflict then either (s_1', r_1') and s_2 conflict or (s_1, r_1) and s_1' conflict.*

As a result of Theorems 4.3 and 4.4, given any process i, in addition to the matches that have PR events in L_i, we will only store matches with wildcard sources, and among such matches, we will only store the most recent match for each tag value (including the wildcard value). Furthermore, in order to speed the search for a conflicting match (s_1, r_1), we will store the match (s_1, r_1) in a balanced search tree $T_{proc(r_1)}$ indexed according to $tag(r_1)$. Given any PS s_2, we will only have to test at most two matches in the search tree $T_{dest(s_2)}$, one with the wildcard tag and one with tag equal to $tag(s_2)$.

4.5 Analysis

We first consider the time required to maintain the logical clocks. Given each new event or match x that is read by the program, A^x is a function of at most $O(N_x)$ other clocks. These clocks can be located in $O(N_x)$ time, and all N_p components of each clock must be examined, so the time required to calculate A^x is $O(N_p N_x)$. When a new match (s, r) is read, the clock $D^{(s,r)}$ is initialized in $O(N_p)$ time. Then when a new event x is read, at most $O(N_x)$ Type D clocks must be updated. These clocks can be found in $O(N_x^2)$ time by using a simple marking algorithm to calculate the R_m^* relation. Once these clocks have been found, their Type D clocks can each be updated in constant time, so the time required to update Type D clocks given a new event x is $O(N_x^2)$. Thus the total time spent calculating clocks for all events and matches is $O(N_e N_p N_x + N_e N_x^2)$.

Given a PS s_2, in order to test if there exists a match (s_1, r_1) such that (s_1, r_1) and s_2 conflict, at most $O(N_x)$ matches with PR events in $L_{dest(s_2)}$ have to be tested, along with at most two matches

in $T_{dest(s_2)}$. Because $T_{dest(s_2)}$ contains at most N_t matches, all of the possibly conflicting matches can be found in $O(N_x + \log N_t)$ time. Each of these matches can be tested for a conflict in $O(N_p)$ time, so the total time for testing conflicts with a PS s_2 is $O(N_p N_x + \log N_t)$, and the time for testing for conflicts with all PS events is $O(N_e N_p N_x + N_e \log N_t)$. We must also maintain the necessary pointers and balanced search trees, but these operations do not dominate the running time. As a result, the entire race detection algorithm requires $O(N_e N_p N_x + N_e N_x^2 + N_e \log N_t)$ time. This is a significant improvement over the naive $O(N_e^2 N_s^2)$ time off-line algorithm based on Theorem 2.2.

Although our algorithm uses logical clocks of length N_p, the fact that older events and matches can be deleted greatly limits the storage required. In particular, only $O(N_p N_x)$ events and $O(N_p N_x + N_p N_t)$ matches (N_x per L_i and N_t per T_i) must be stored. Because each event and each match requires $O(N_p)$ memory, the entire algorithm uses $O(N_p^2 N_x + N_p^2 N_t)$ memory.

5 Concluding Remarks

This paper described an algorithm that detects race conditions in program executions which use blocking and nonblocking send and receive primitives. The key to the algorithm's efficiency is the use of two new types of logical clocks for testing the "happened-before" relation for blocking and nonblocking sends and receives.

In the future we plan to create a tool that will automatically determine if an execution of a message-passing program is portable and repeatable. We have shown that any execution (with locally deterministic events) is portable and repeatable if and only if it is free from race conditions and it does not have any system buffer dependencies [7]. Thus we will use the race detection algorithm presented here, in conjunction with a test for system buffer dependencies, to determine the portability and repeatability of the execution.

References

[1] D. Callahan, K. Kennedy and J. Subhlok, "Analysis of Event Synchronization in a Parallel Programming Tool", in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, pp. 21–30, 1990.

[2] J-D. Choi and S. Min, "Race Frontier: Reproducing Data Races in Parallel-Program Debugging", in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, pp. 145–154, 1991.

[3] T. Cormen, C. Leiserson and R. Rivest, "Introduction to Algorithms", The MIT Press, Cambridge, MA, 1990.

[4] R. Curtis and L. Wittie, "BugNet: A Debugging System for Parallel Programming Environments", in *Proc. 3rd Intl. Conf. on Distributed Computing Systems*, pp. 394-399, Oct. 1982.

[5] R. Cypher, "Message-Passing Models for Blocking and Nonblocking Communication", *DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation*, pp. 274–279, Tech. Rep. 93-87, DIMACS, Sept. 1993.

[6] R. Cypher and E. Leu, "The Semantics of Blocking and Nonblocking Send and Receive Primitives", in *Proc. 8th IEEE Intl. Parallel Processing Symp.*, pp. 729-735, 1994.

[7] R. Cypher and E. Leu, "Repeatable and Portable Message-Passing Programs", in *Proc. ACM Symp. on Principles of Distributed Computing*, pp. 22-31, 1994.

[8] S. Damodaran-Kamal and J. Francioni, "Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs", in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 118–128, 1993.

[9] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection", in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, pp. 1–10, 1990.

[10] J. Dongarra, R. Hempel, A. Hey and D. Walker, "A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment", ORNL Tech. Rep., ORNL/TM-12231, June 1993.

[11] P. Emrath, S. Ghosh and D. Padua, "Event Synchronization Analysis for Debugging Parallel Programs", in *Proc. Supercomputing '89*, pp. 580–588, Nov. 1989.

[12] C. Fidge, "Logical Time in Distributed Computing Systems", *IEEE Computer*, pp. 28–33, Aug. 1991.

[13] D. Frye, R. Bryant, H. Ho, R. Lawrence and M. Snir, "An External User Interface for Scalable Parallel Systems", Tech. Rep., IBM Highly Parallel Supercomputing Systems Lab., Nov. 1992.

[14] D. Helmbold, C. McDowell and J-Z. Wang, "Determining Possible Event Orders by Analyzing Sequential Traces", *IEEE Trans. on Parallel and Distributed Systems*, 4(7), pp. 827–840, 1993.

[15] M. Hurfin, N. Plouzeau and M. Raynal, "EREBUS: A Debugger for Asynchronous Distributed Systems", In *Proc. 3rd Intl. IEEE Workshop on Future Trends in Distributed Computing Systems*, pp. 93-98, Apr. 1992.

[16] Intel Corp., "Intel iPSC/860 Programmer's Reference Manual".

[17] Y-K. Jun and K. Koh, "On-the-fly Detection of Access Anomalies in Nested Parallel Loops", in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 107–117, 1993.

[18] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, 21(7), pp. 558-565, 1978.

[19] T. Leblanc and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, C-36(4), pp. 471-482, 1987.

[20] E. Leu, A. Schiper and A. Zramdini, "Efficient Execution Replay Technique for Distributed Memory Architectures", in *Proc. 2nd European Distributed Memory Computing Conference*, LNCS 487, Springer-Verlag, pp. 315-324, Apr. 1991.

[21] F. Mattern, "Virtual Time and Global States of Distributed Systems", in *Parallel and Distributed Algorithms*, M. Cosnard and P. Quinton, eds., North-Holland, Amsterdam, pp. 215–226, 1989.

[22] J. Mellor-Crummey, "Compile-Time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs", in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 129–139, 1993.

[23] Message Passing Interface Forum, "Document for a Standard Message-Passing Interface (Draft)", Apr. 1994.

[24] R. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs", in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1–11, 1993.

[25] R. Netzer and B. Miller, "Improving the Accuracy of Data Race Detection", in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, pp. 133–144, 1991.

[26] R. Netzer and B. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs", in *Proc. Supercomputing '92*, pp. 502-511, Nov. 1992.

[27] Thinking Machines Corp., "CMMD Reference Manual, version 3.0", May 1993.

Appendix

Theorem 3.1 *Given any event a and any event or match b where $a \neq b$,*

$$R_h^+(a,b) \Leftrightarrow A^a[proc(a)] \leq A^b[proc(a)].$$

Proof: a) $\Rightarrow$: If $R_h^+(a,b)$, there must exist a sequence of events and/or matches $x_1, x_2, \ldots, x_k$ such that $x_1 = a$, $x_k = b$, and $R_h(x_i, x_{i+1})$ for $1 \leq i < k$. If x_i and x_{i+1} are both events, it follows that either $A^{x_{i+1}}[proc(a)] = A^{x_i}[proc(a)]$ or $A^{x_{i+1}}[proc(a)] = A^{x_i}[proc(a)] + 1$. If one (or both) of x_i and x_{i+1} is a match, it follows that $A^{x_{i+1}}[proc(a)] \geq A^{x_i}[proc(a)]$. Therefore, in any case $A^{x_{i+1}}[proc(a)] \geq A^{x_i}[proc(a)]$, which implies that $A^b[proc(a)] \geq A^a[proc(a)]$.

b) $\Leftarrow$: Assume for the sake of contradiction that $A^a[proc(a)] \leq A^b[proc(a)] \wedge \neg R_h^+(a,b)$. Given any event or match x, let S_x be the set of all events and matches y such that $R_h(y,x)$. Note that for any event or match x, A^x is a function of the Type A clocks of the events and matches in S_x. Given any event or match x, define $inherited(x,a)$ to be an event or match $y \in S_x$ such that for all $z \in S_x$, $A^z[proc(a)] \leq A^y[proc(a)]$. Note that $inherited(x,a)$ is defined for all x such that S_x is nonempty.

Let $x_1, x_2, \ldots, x_k$ be the sequence of events and matches such that $x_1 = b$, S_{x_k} is empty, and $x_{i+1} = inherited(x_i, a)$ for $1 \leq i < k$ (the sequence must be finite because $x_{i+1} = inherited(x_i, a)$ implies $R_h(x_{i+1}, x_i)$). It follows from the definition of Type A clocks that for all i, $1 \leq i < k$, either $A^{x_i}[proc(a)] = A^{x_{i+1}}[proc(a)]$ or $A^{x_i}[proc(a)] = A^{x_{i+1}}[proc(a)] + 1$. Let j be the largest integer less than or equal to k such that $A^{x_j}[proc(a)] = A^a[proc(a)]$ (j must exist because $A^a[proc(a)] \geq 1$ and $A^{x_k}[proc(a)] \leq 1$). Note that either $x_j = x_k$ or there exists an x_{j+1} such that $A^{x_{j+1}}[proc(a)] = A^a[proc(a)] - 1$. If $x_j = x_k$, because $A^a[proc(a)] \geq 1$ it follows that $A^{x_j}[proc(a)] = 1$ and x_j is the first event in process $proc(a)$, so it follows that $x_j = a$. On the other hand, if $A^{x_{j+1}}[proc(a)] = A^a[proc(a)] - 1$, it follows from the definition of Type A clocks that $proc(x_j) = proc(a)$. Because $A^{x_j}[proc(a)] = A^a[proc(a)]$, it follows that $x_j = a$. Therefore, in either case $x_j = a$, which implies that $R_h^+(a,b)$, which is a contradiction. □

Theorem 3.3 *Given any match (s,r) and any event x, $R_h^+((s,r),x) \Leftrightarrow \exists j$ such that $C^{(s,r)}[j] \leq A^x[j]$.*

Proof: a) $\Rightarrow$: If $R_h^+((s,r),x)$, it follows that there exists a sequence of events and matches $y_1, y_2, \ldots, y_k$ such that $y_1 = (s,r)$, $y_k = x$, and for all i, $1 \leq i < k$, $R_h(y_i, y_{i+1})$. Let h be the smallest integer such that y_h is an event. Note that $R_h(y_{h-1}, y_h)$ where y_{h-1} is a match and y_h is an event. Also, note that either $(s,r) = y_{h-1}$ or $R_m^+((s,r), y_{h-1})$. Therefore, it follows from the definition of Type C clocks that $C^{(s,r)}[proc(y_h)] \leq A^{y_h}[proc(y_h)]$. Also, note that either $y_h = x$ or $R_h^+(y_h, x)$. Therefore, it follows from Theorem 3.1 that $A^{y_h}[proc(y_h)] \leq A^x[proc(y_h)]$. As a result, $C^{(s,r)}[proc(y_h)] \leq A^x[proc(y_h)]$.

b) $\Leftarrow$: Assume for the sake of contradiction that there exists a match (s,r), an event x, and a process j such that $C^{(s,r)}[j] \leq A^x[j] \wedge \neg R_h^+((s,r),x)$. Let S be the set of all events z in process j such that there exists a match (s',r') where $R_m^*((s,r),(s',r')) \wedge R_h((s',r'),z)$. It follows from the definition of Type C clocks that there exists an event $z \in S$ such that $C^{(s,r)}[j] = A^z[j]$ (z must exist because $C^{(s,r)}[j] \neq \infty$). Because $C^{(s,r)}[j] \leq A^x[j]$, it follows that $A^z[j] \leq A^x[j]$, and because $proc(z) = j$, it follows from Theorem 3.1 that either $z = x$ or $R_h^+(z,x)$. However, $R_h^+((s,r),z)$, so $R_h^+((s,r),x)$, which is a contradition. □