Detecting Data Races by Analyzing Sequential Traces

David P. Helmbold
Charles E. McDowell
Jian-Zhong Wang

90-57
October 17, 1990

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

One of the fundamental problems encountered when debugging a parallel program is determining the potential race conditions in the program. A race condition exists when multiple tasks access shared data in an unconstrained order and at least one of the accesses is a write operation. The program's behavior can be unpredictable when race conditions are present. This paper describes techniques which automatically detect data races in parallel programs by analyzing program traces.

We view a program execution as a partial ordering of events, and define which executions are consistent with a given trace. In general, it is not possible to determine which of the consistent executions occurred. Therefore we introduce the notion of "safe orderings" between events which are guaranteed to hold in every execution which is consistent with the trace. The main result of the paper is a series of algorithms which determine many of the "safe orderings". An algorithm is also presented to distinguish unordered sequential events from concurrent events.

A working trace analyzer has been implemented. The trace analyzer can report various data races in parallel programs by finding unordered pairs of events and variable access conflicts.

Keywords: data race, time vector, program trace, parallel programming, debugging, distributed systems

1 Introduction

Writing and debugging a parallel program is, in general, more difficult than writing and debugging a sequential program. A major reason for this difficulty is the need for explicit synchronization between the tasks in a parallel program. A program with errors in synchronization will often be *non-determinate*, i.e., generate different results even when started with exactly the same inputs. In a parallel program, nondeterminism often introduces unexpected program behavior, making the debugging process extremely difficult.

Unwanted non-determinate behavior of parallel programs often starts with a *data race*. One of the fundamental problems encountered when debugging a parallel program is locating the potential data races in the program. A race occurs when two or more parallel tasks access some shared variable in an unspecified order, and at least one of the accesses is a write access. For example, one task may attempt to write to a memory location while a second task is reading from that memory location. The behavior of the second task may differ dramatically depending on whether it reads the new value or the old one. Notice that races include both accesses that may occur "at the same time" and accesses that must occur sequentially but where the order is unspecified (e.g. accesses protected by a lock).

One approach to determining potential races is based on computing all of the reachable concurrency states of the program [McD89, Tay84]. The major disadvantage of this approach is that the number of concurrency states may become prohibitively large. Another approach to determining potential races is based on analyzing a trace from an execution of the program [EP88, MC88, EGP89, NM89, HMW90, HMW91]. One advantage of trace analysis is that it is much less expensive computationally than the known static structure analysis methods. Another advantage is that trace analysis may avoid the problem of reporting (potentially too many) spurious data races by structure analysis, due to infeasible paths in the program. However, the races detectable by analysis of event histories may depend on the program input data used to generate the trace. Furthermore, since races introduce nondeterminism, a data race may hide other data races from the trace analyzer [AP87]. Nevertheless, this later approach can provide important information to help in debugging parallel programs and is the subject of this paper.

When debugging a parallel program, the first step is to determine the order and concurrency relationships among the operations performed by the program. When examining a trace, the problem becomes distinguishing the ordered event pairs from the unordered, potentially concurrent, event pairs. In order to perform the final data race analysis, it must be possible to determine from a trace what shared objects are referenced between any two synchronization events. This can be done by generating an event whenever an object is accessed. Alternatively, each synchronization event could include the source line number of the statement generating the event. From the source line numbers the path between two adjacent events can be determined and the variables referenced along the path can be computed [McD89].

A *trace* (also called an *event history*) is a linear list or total ordering of the events performed during an execution of the program. For our purposes, the trace reflects

only one of the orders in which the events could have occurred. A more restrictive definition that is difficult to achieve in practice[1] would be for a trace to specify the exact order in which the events did occur. Since traces are only approximations of executions, there are usually several executions that are consistent with a given trace. What we want to compute is those pairs of events that occur in the same order in every execution which is consistent with the trace. These *must occur* orderings can be viewed as a partial order. If the partial order contains all orderings that must occur, then a pair of events not ordered by this "*must occur*" partial ordering are unordered in some consistent execution or consistent executions exist where the two events are executed in both possible orders.

If the trace contains explicit synchronization such as rendezvous between tasks t and t' or task t forks into tasks t_1, t_2, and t_3, then it is easy to determine these pairs of events that must occur in a particular order. However, if the trace contains anonymous synchronization (e.g. semaphores, locks, signals) then determining whether or not two events occur in a particular order in every consistent execution is much more difficult[2].

Many parallel systems (e.g. [IBM88]) provide facilities for recording important events during the execution of parallel programs. By limiting the debugger's activity, the *probe effect* should be reduced. The recorded information can be analyzed following the program's execution.

The next section contains definitions and description of our basic model involving counting semaphores. Our algorithms for this basic model are described and analyzed in Section 3. We have implemented a version of our algorithms for the post/wait style synchronization used in IBM's Parallel Fortran. Our implementation and the necessary modifications to our algorithms are described in Sections 4 and 5. In Section 6 we survey some related work. Finally, Section 7 contains conclusions and a brief summary of our results.

2 Description of the Model

During an execution, a parallel program initiates a finite set of *tasks* $\{T_1, \cdots, T_n\}$. These tasks perform synchronization and computation operations, including computation on shared data[3]. A program *trace* (or event history) H is a linearly ordered sequence of events generated by a program execution. An *event* is a traced significant program step, which generally includes the synchronization operations. Each task T_i is a sequential entity characterized by a local sequence H_i of events[4].

[1] When events occur concurrently, it is impossible for a totally ordered trace to accurately reflect the event orderings.

[2] In fact, the problem of determining all "must occur" orderings between events in a counting semaphore model has been shown to be co-NP-hard [NM90].

[3] Although operations on shared data can be used for synchronization [Dij65], we only consider explicit synchronization operations as capable of generating synchronization events.

[4] Appearance of an event indicates that the event has completed.

Different tasks may perform operations concurrently. We assume, for convenience, that each task has a unique identifier.

In our basic model, programs synchronize using counting semaphores (initialized to zero). Two operations, **P** and **V**, are defined for each semaphore. In this paper, we use the more mnemonic wait and signal to represent the **P** and **V** operations respectively. Therefore, each synchronization event is a tuple containing: the operation completed (wait or signal), the affected semaphore, and the id of the task that performed the operation.

Many other kinds of synchronization operations can be simulated with counting semaphores. Consider, for example, the event "*init task t*" which creates a new task t and the event "*await task t*" which blocks the running task until task t has terminated. Given a trace containing these events, we can create an equivalent trace containing only semaphore events.

In each execution every wait event (blocking event) has a corresponding signal event (enabling event). We use this correspondence to define a partial order representing that execution.

Definition 1: *An* execution *of a parallel program is an irreflexive partial ordering of the events performed. This partial order, called an* event history graph (EHG)*, is the transitive closure*[5] *of two types of edges: (1) edges from each event to the next event performed by the same task and (2) edges from each enabling synchronization event to its corresponding blocking synchronization event (or events).*

An EHG is a physical representation of the concept of an execution and we will use the two synonymously. Although other partial orderings on the events can (and will) be defined, we reserve the term "EHG" for those partial orderings representing possible executions of the program. The relation defined by an EHG is called the *happened before* relation and is denoted with the symbol $\rightarrow$. Our definition of "happened before" is consistent with that of Lamport[Lam78].

Definition 2: *Consider an EHG and two distinct events* e, e'*. If* $e \not\rightarrow e'$ *and* $e' \not\rightarrow e$ *then events* e *and* e' *are* concurrent *in that EHG, and thus can happen at the same time in the execution represented by the EHG.*

Definition 3: *A* trace *of an execution is an interleaving of the local sequences of events* H_i *for* $1 \leq i \leq n$ *where for every prefix of the trace and every semaphore S, the prefix contains at least as many signal(S) events as wait(S) events.*

A single execution usually has many possible traces. Similarly, most traces could have been generated by any one of a number of executions. (Figures 2.1(a) and 2.1(b) show the EHGs for two different executions capable of generating the same trace).

Definition 4: *An EHG or execution is* consistent *with a trace if the local sequences of trace events* H_i *for each task* $1 \leq i \leq n$ *is preserved by the partial ordering of the EHG.*

[5] Although EHGs are transitively closed, we usually omit the transitive edges when drawing them.

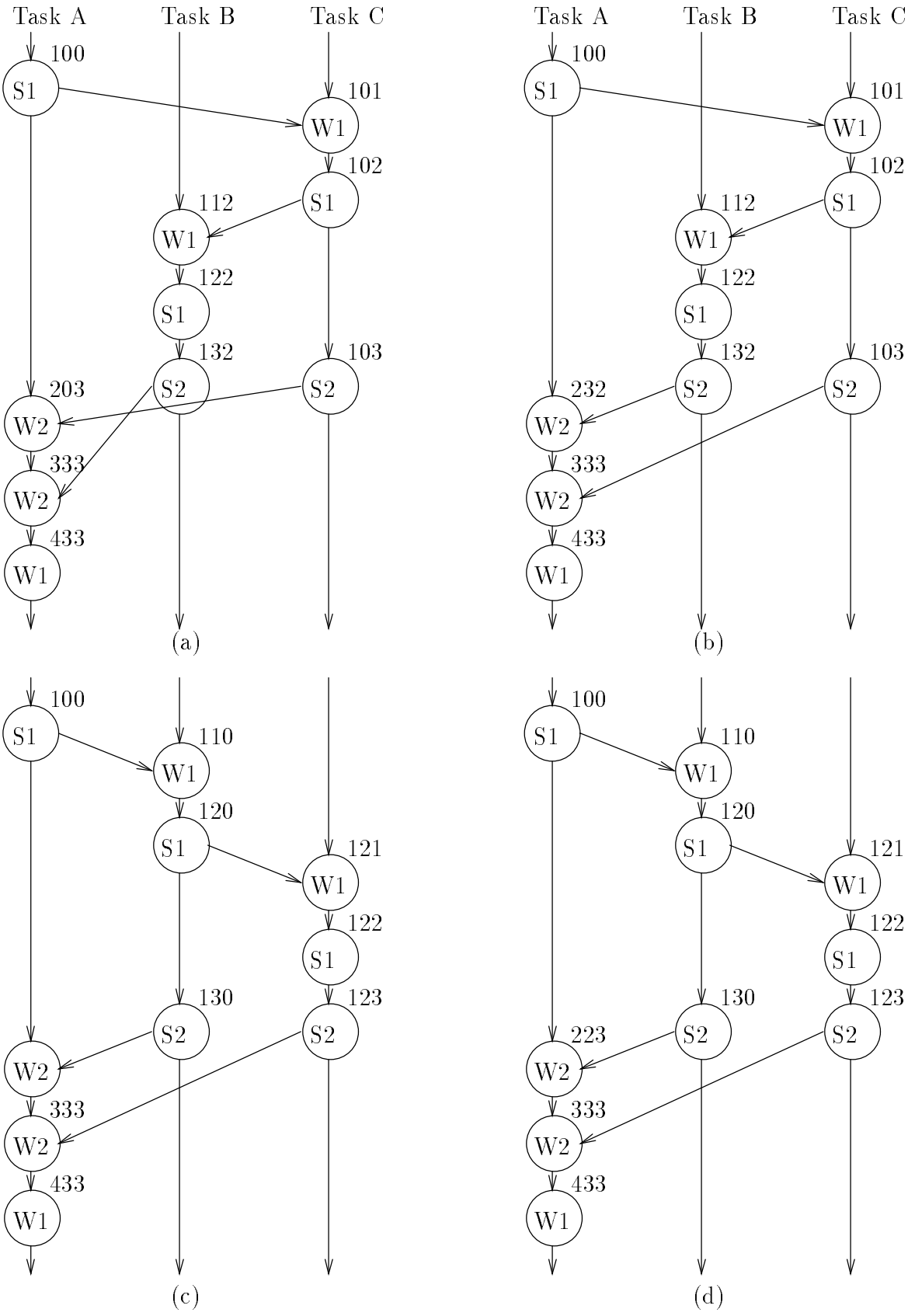

Figure 2.1: Trace, Executions, and Time Vectors

This notion of consistency is very important. Intuitively, an EHG is consistent with every trace where the individual tasks appear to have done the same things.

For example, consider the trace H = {AS1, CW1, CS1, CS2, BW1, BS1, BS2, AW2, AW2, AW1}. Event AS1 means task A performs a signal(S_1), AW1 means task A performs wait(S_1) etc. Figure 2.1 shows the four EHGs which are consistent with this trace.

Given the sequence of events forming a trace history, we want to analyze the trace and report every data race which can happen in an execution whose EHG is consistent with the trace. A race may exist if there is no way provided in the program to force the execution order between two events. Our tool detects concurrent or unordered read/write and write/write accesses to the same variable data races.

Definition 5: *Given a trace, the symbol "$\|$" is used to represent the* may happen concurrently *relationship between events. Two events e and e' are concurrent with respect to the trace (i.e. $e \| e'$) if they are concurrent in some EHG which is consistent with the trace.*

Definition 6: *Given a trace, the symbol "$\prec$" is used to represent the* must happen before *relationship between events. If $e \prec e'$, where e and e' are events, then event e will happen before e' in all executions that are consistent with the given trace. Events e and e' are* ordered *if $e \prec e'$ or $e' \prec e$, otherwise, they are* unordered.

Concurrent events are always unordered, but unordered events need not be concurrent. For example, the events BW1 and CW1 in Figure 2.1 are unordered but not concurrent.

Notice that $e \prec e'$ is usually different from the $e \rightarrow e'$ relation for any particular EHG. The former relation tells us that e *must happen before* e' in all of the executions consistent with the trace being analyzed, while the later says that e happened before e' in the particular execution represented by the EHG. If $e \prec e'$ then $e \rightarrow e'$ in every consistent EHG. However, the fact that $e \rightarrow e'$ in some consistent EHG does not necessarily mean that $e \prec e'$.

In Figure 2.1, CS1 $\rightarrow$ BW1 in EHG (a). However, in EHG (c), BS1 $\rightarrow$ CW1, and BW1 $\rightarrow$ CS1 by transitivity. Therefore, BW1 and CS1 are unordered. Event AS1 happens before BW1 and CW1 in every EHG consistent with the trace, therefore AS1 $\prec$ BW1 and AS1 $\prec$ CW1. There is no order relation between events CS2 and BW1 in execution (a). Therefore, they can happen concurrently in some execution consistent with the trace and CS2 $\|$ BW1.

Definition 7: *A partial ordering R on the events is a* safe *order relation with respect to a trace if $e_i \; R \; e_j \Rightarrow e_i \prec e_j$. If R is not safe, then R is* unsafe.

Given a trace, the safe order relations are subsets of the must happen before relationship for that trace. In general, the consistent EHGs are never safe, because they contain too much information about their particular executions.

2.1 Virtual Time

Since a linearly ordered representation of time is not always adequate for debugging parallel programs, we use time vectors to represent a partial order on the events. In our trace analysis, each event is assigned a vector of timestamps. The ordered event pairs and unordered event pairs can be easily distinguished by comparing these time vectors.

The time vectors we compute in this paper are an extension of the time vectors of Fidge [Fid88] and Mattern [Mat88]. There, each task T_i has a clock C_i which is a vector of length n, where n is the total number of tasks.[6] Each task T_i has its own vector component $C_i[i]$ which guarantees a strict linear ordering of events occurring in that task. A local event counter which is incremented each time an event occurs in the task can be used as the local clock. The other components get updated when the task synchronizes with other tasks.

The time vectors assigned to events represent a partial ordering of the events. Intuitively, event e_1 precedes another event e_2 in the partial order represented if every component of e_1's time vector τ_1 is less than or equal to the corresponding component in e_2's time vector τ_2. Events e_1 and e_2 are unrelated in the partial order when both some component of τ_1 is greater than the corresponding component of τ_2, and some other component of τ_2 is greater than the corresponding component in τ_1.

Definition 8: *For any two time vectors τ_1, τ_2 in Z^n*

1. $\tau_1 \leq \tau_2 \iff \forall i(\tau_1[i] \leq \tau_2[i])$
2. $\tau_1 < \tau_2 \iff \tau_1 \leq \tau_2$ *and* $\tau_1 \neq \tau_2$
3. $\tau_1 \parallel \tau_2 \iff \neg(\tau_1 < \tau_2)$ *and* $\neg(\tau_2 < \tau_1)$.

Time vector τ_1 is earlier than time vector τ_2 if and only if they are different and every component of τ_1 is less than or equal to the corresponding component of τ_2. For any two distinct time vectors τ_1 and τ_2, if neither τ_1 is earlier than τ_2 nor τ_2 is earlier than τ_1, then we define τ_1 and τ_2 to be *unordered*. Note that here we define "$\parallel$" for time vectors, whereas Definition 5 defined "$\parallel$" for events. However, when the time vectors represent the must happen before relationships for some trace, then the two definitions of "$\parallel$" coincide.

Definition 9: *For any m time vectors $\tau_1, \ldots, \tau_m$ of Z^n*

- $\overline{\min}_k(\tau_1, \ldots, \tau_m), k > 0$ *is the vector of Z^n whose ith component is the k^{th} smallest element of $(\tau_1[i], \ldots, \tau_m[i])$,*
- $\overline{\min}_0(\tau_1, \ldots, \tau_m)$ *is the vector of Z^n with zeros everywhere, and*
- $\overline{\max}(\tau_1, \ldots, \tau_m)$ *is a vector of Z^n whose ith component is* $\max(\tau_1[i], \ldots, \tau_m[i])$.

[6] We use an integer valued clock in our discussion although a real number valued clock can also be used.

As an example, $\overline{\min}_3([1,2],[1,3],[2,4],[2,5],[3,2])$ is $[2,3]$. We often call $\overline{\min}_k(\tau_1,\ldots,\tau_m)$ the k^{th} component-wise minimum of $\tau_1,\ldots,\tau_m$, and $\overline{\max}(\tau_1,\ldots,\tau_m)$ the component-wise maximum of $\tau_1,\ldots,\tau_m$.

The following algorithm (derived from [Mat88, Fid88]) computes time vectors for the events in an EHG. This algorithm requires the correspondence between signal and wait events. The time vectors produced reflect the happened before partial order for that EHG.

Definition 10: *For an event $e \in H_i$, e^p is the previous event performed by the same task T_i if such an event exists.*

Definition 11: *For an event $e \in H_i$, $\tau^{\#}(e)$ is the time vector containing the local event count for e in the ith component and zeros elsewhere.*

Algorithm 1: Given the correspondence between signal and wait events for an EHG, each event e is assigned a time vector, $\tau(e)$, as follows:

$$\tau(e) \quad = \quad \overline{\max}(v_t, v_s, \tau^{\#}(e))$$

where

$$v_t \quad = \quad \begin{cases} \tau(e^p) & \text{if there is a previous event} \\ & \text{performed by the same task} \\ \text{the 0 vector} & \text{otherwise} \end{cases}$$

$$v_s \quad = \quad \begin{cases} \tau(\hat{e}) & \text{if } e \text{ is a wait event and} \\ & \hat{e} \text{ is the corresponding signal} \\ \text{the 0 vector} & \text{if } e \text{ is not a wait event} \end{cases}$$

End Algorithm 1.

Now, $\tau(e) < \tau(e')$ iff e happened before e' in the EHG; and $\tau(e) \parallel \tau(e')$ iff e and e' are concurrent in that EHG.

In Section 3 we consider the counting semaphore style synchronization model. The algorithms presented there take a linearly ordered trace containing counting semaphore synchronization, and compute a partial order containing only *must happen before* type orderings. If event e_1 has an earlier time vector than e_2 in the computed partial order, then e_1 must happen before e_2 in *all* executions that are consistent with the given trace. By comparing their final time vectors, we can distinguish many *ordered* events from the *unordered* events.

3 Analyzing Traces with Anonymous Synchronization

The algorithms presented in this section have three phases: "initialize", "rewind", and "expand". The partial order resulting from the initialization phase is similar to that computed by the algorithm of [Fid88]. This partial order is a consistent EHG, so it is likely to be an unsafe order relation. The result of the rewind step is a partial order that is a safe order relation. Unfortunately, it is an overly conservative safe

order relation, since many of the safe ordering edges may have been lost during the rewinding procedure. The expanding process is used to add additional safe ordering edges to the partial order.

Our goal is a set of time vectors which can be used to distinguish ordered events from unordered and potentially concurrent events[7]. The problem of calculating all safe order relations has been shown to be co-NP-hard by Netzer and Miller [NM90]. What we present is a good approximation of the problem. Our methods will find many, if not all, of the safe orderings in most parallel programs.

3.1 Initializing the Time Vectors

Before giving the algorithm for computing the initial time vectors, we define a canonical EHG that will be used to verify the "correctness" of the time vectors. This canonical EHG simply matches the ith wait event on semaphore S with the ith signal event on the same semaphore.

Definition 12: *Given a trace H with the total ordering of events, $<_H$, the partial order $\rightarrow$ corresponding to the* canonical EHG *is constructed by selecting and taking the transitive closure of the following subrelation of $<_H$.*

- *If e_i and e_j are two events from the same task and $e_i <_H e_j$ then $e_i \rightarrow e_j$.*
- *If e_i and e_j are the kth signal and wait events respectively on the same semaphore, then $e_i \rightarrow e_j$.*

Unless indicated otherwise, "$\rightarrow$" represents the happened before relation for the canonical EHG in the remainder of this section.

Algorithm 2: To compute initial time vectors, $\tau(e)$, from a trace H use algorithm 1 with the following modifications[8].

- The kth wait event on semaphore S (in trace order) corresponds to the kth signal event on S.
- The events are assigned time vectors in the order they appear in the trace.

End Algorithm 2.

For the given trace, Figure 2.1(a) shows the result of the initialization procedure.

The time vectors computed for the canonical execution have the following properties:

Property 1: *If e and $\hat{e}$ are two events in the same task T_i and e occurred before $\hat{e}$ in the trace, then $e \rightarrow \hat{e}$ and $\tau(e) < \tau(\hat{e})$.*

---

[7]Given a specific input and trace, there may be executions on that input whose EHGs are not consistent with the trace, however, any such execution will contain a race if and only if a race occurred in the execution that generated the trace [AP87].

[8]Here we assume that matching up the i^{th} signal with the i^{th} wait in the trace gives a legal execution. If that is not the case, then the parings of any legal execution could be used. However, determining if there are any EHGs consistent with an arbitrary list of events is an NP-complete problem.

Property 2: *If e and $\hat{e}$ are the corresponding signal and wait pair (the kth signal and the kth wait on the same semaphore S in the trace), then $e \rightarrow \hat{e}$ and $\tau(e) < \tau(\hat{e})$.*

Property 3: *At any point in the trace, the maximum value of any time vector component is the number of events performed up to that point by the task associated with that component.*

Given the correspondence between signal and wait events in some EHG, events can be assigned time vectors by using Algorithm 1. Mattern [Mat88] has shown that the resulting time vectors correctly represent the relation $\rightarrow$ for that EHG. Therefore, the initial time vectors, τ, correctly represent the happened before relation for the canonical execution.

Theorem 1: *For any pair of distinct events $e_i \in H_i$ and $e \in H$,*

$$\tau(e_i)[i] \leq \tau(e)[i] \Longleftrightarrow e_i \rightarrow e.$$

See [Wan90] for detailed proofs of the theorems appearing in this paper.

Corollary 1: *For any two distinct events $e \in H_i$, $\hat{e} \in H_j$, $i \neq j$, if $\tau(e)[i] > \tau(\hat{e})[i]$ and $\tau(\hat{e})[j] > \tau(e)[j]$, then e and $\hat{e}$ are concurrent in the canonical execution.*

The initialization process creates an EHG consistent with the given trace. Unfortunately, this partial order is (in general) an unsafe order relation. The correspondence between signals and waits in the canonical EHG need not hold for other execution(s) capable of generating the trace. Even when $\tau(e) < \tau(\hat{e})$ we cannot say *e must happen before $\hat{e}$.*

3.2 Rewinding the Time Vectors

The result of the initialize step in the previous section is an unsafe order relation. It is unsafe because we assumed that the kth signal event for a particular semaphore was the one allowing the kth wait event to precede. The next step is to rewind the time vectors to account for the fact that any signal event might be the one that allowed any wait event on the same semaphore to complete. We use $\tau'(e)$ to represent the new time vector assigned to event e during and after the rewinding process. Initially τ' is the same as τ.

Suppose e is a wait event, and e_1 and e_2 are the only two signal events which could have caused e to complete. In this case, we only know that either e_1 or e_2 must have happened before e. The trace might be in any of the forms:

$$\ldots, e_1, \ldots, e, \ldots, e_2, \ldots;$$
$$\ldots, e_2, \ldots, e, \ldots, e_1, \ldots;$$
$$\ldots, e_1, \ldots, e_2, \ldots, e, \ldots; \quad \text{or}$$
$$\ldots, e_2, \ldots, e_1, \ldots, e, \ldots.$$

However, we can conclude that those events preceding both e_1 and e_2 must also occur before e. Formally if $e_a \prec e_1$ and $e_a \prec e_2$ then $e_a \prec e$. The rewind step defined below uses this fact to obtain a safe order relation.

Algorithm 3: (Rewind)
Initially, $\forall e \in H, \tau'(e) = \tau(e)$.
Repeat the following procedure until no further changes are possible.
For all events $e \in H$, let

$$\tau'(e) = \overline{\max}(\tau'(e^p), \tau^{\#}(e), v_s)$$

where if e is a wait event on semaphore S:

$$v_s = \overline{\min}(\tau'(e_1^s), \ldots, \tau'(e_k^s))$$

where $e_1^s \ldots e_k^s$ are all the signals on S;
otherwise v_s is the 0 vector.
End Algorithm 3.

Observe that the only difference between Algorithm 3 and Algorithm 2 (used to compute τ) is that for wait events in Algorithm 3, v_s is the minimum of a set of time vectors, which includes the time vector used for v_s in computing τ. Therefore the values of τ' will only get smaller as Algorithm 3 executes.

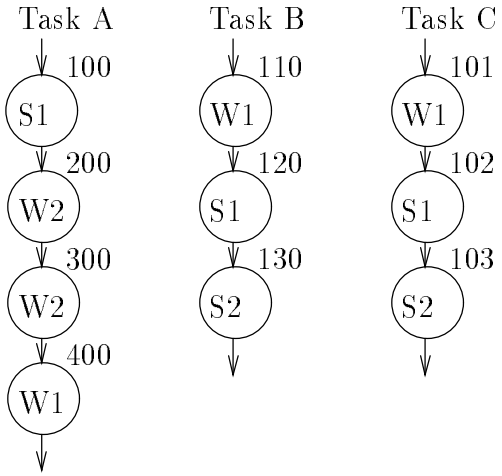

Figure 3.1: Rewinding the Time Vectors

After rewinding, we have a partial order that is a safe order relation (although probably not an EHG). If event e_i has an earlier time vector than e, then e_i happens before e in all executions that are consistent with the given trace.

Theorem 2: *Algorithm 3 generates only safe order relations, i.e., for any two events* $e_1, e_2 \in H$*:*

$$\tau'(e_1) < \tau'(e_2) \Rightarrow e_1 \prec e_2.$$

The rewinding process is based on the fact that any signal event might enable any wait event on the same semaphore. We may have lost some safe ordering edges during rewinding. As an example, the τ' time vectors in Figure 3.1 indicate that the two W2 events and the W1 event in task A may happen concurrently with any of the

events in tasks B and C. However, it is obvious that the W1 in task A must follow the two S1 events in tasks B and C, and the second W2 in task A has to wait until all of the events in B and C have occurred. The final step in the algorithm restores some of the edges lost during the rewinding procedure.

3.3 Expanding the Safe Order Relation

The result of the rewind step is a partial order that is a safe order relation. Unfortunately, it is an overly conservative safe order relation, since some of the safe ordering edges may have been lost during the rewinding procedure. We now undertake a process to add additional safe ordering edges into the partial order. The partial order resulting from this process will be represented by the time vectors $\hat{\tau}(e)$. Initially, $\hat{\tau}(e) = \tau'(e)$.

Suppose e is a wait event on some semaphore S, and there are k other wait events on S which must happen before e in every consistent execution. In this case, at least $k+1$ signal events on S are needed in order for e to proceed. We will show that this fact can be used to get more safe ordering edges. As an extreme example, given a trace $H = \{AS, BW, AS, AS, BW, BW\}$, Figure 3.2(a) shows the result of rewinding. It appears that the only inter-task safe ordering is that the first signal in task A must happen before all the wait events in task B. However, the second signal in A must happen before the second wait in B, and the third signal must happen before the third wait, as shown in Figure 3.2(b).

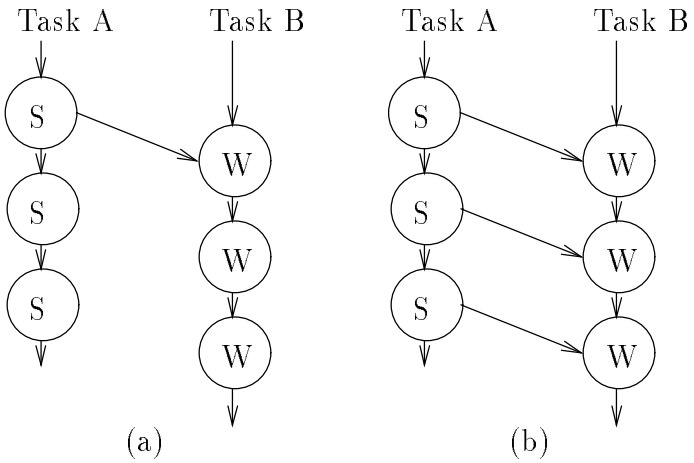

Figure 3.2: Safe Ordering

Additional safe ordering edges can be found based on the following observation. If some wait event e is known to follow a set of k other wait events, then there is a demand for $k+1$ signals preceding e. If some signal used to meet this demand is itself preceded by a wait not in the set, then this latter wait will increase the demand for signals to $k+2$. Therefore the demand satisfied and additional demand created by including a signal can cancel. When this happens we say that the signal is shadowed.

In the example shown in Figure 3.3, the signal event CS1 is preceded by a wait event CW1. For the second BW1, from the current time vectors, we know that there is a wait event (the first BW1) preceding it. Therefore, at least two signal(S1) events need to precede the second BW1. CS1 could be one of them, but if this is the case, then at least three signal(S1) events are needed to allow the second BW1 event to precede. In any consistent execution, the second BW1 must happen after at least two signal(S1) events other than CS1. Similarly, the first BW1 must be preceded by at least one signal(S1) which is not CS1. We say that the signal event CS1 is shadowed by CW1 with respect to the wait events performed by task B.

Definition 13: *Let $e \in H_i$ be a wait event and $e_s \in H_j$ be a signal event on the same semaphore S where $\hat{\tau}(e) \parallel \hat{\tau}(e_s)$. Let $H(e, e_s)$ be the subsequence of H_j containing those events e_j where $e_j \prec e_s$ and $\hat{\tau}(e_j) \parallel \hat{\tau}(e)$. If any suffix of $H(e, e_s)$ contains more wait events on S than signal events on S, then the signal event e_s is* shadowed *with respect to e.*

Definition 14: *Let $H'(e, e_s)$ be the shortest suffix of $H(e, e_s)$ which contains more wait events than signal events on S, and let e_w be the first event of $H'(e, e_s)$. We say e_s is shadowed by event e_w with respect to e.*

Lemma 1: *Given a wait event e and a signal event e_s on the same semaphore S, if e_s is shadowed by some event e_w with respect to e then*

- *event e_w is a wait event on semaphore S,*
- *the subsequence between e_w and e_s (in the same task) contains as many signal events as wait events on semaphore S,*
- *the event e_w, which shadows e_s with respect to e, is unique. We define e_w to be the shadowing wait event corresponding to e_s, and*
- *the correspondence between shadowed signal and shadowing wait is one to one.*

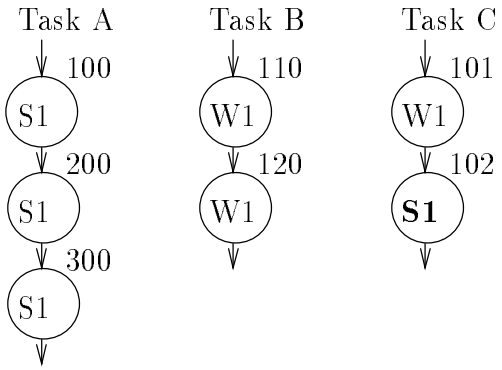

Figure 3.3: Shadowed Signal Event

In the example shown in Figure 3.3, the signal event CS1 is shadowed by CW1 with respect to the two wait events performed by task B.

Algorithm 4 is based on the following observation. If e is a wait event on semaphore S and k other wait events on S must happen before e, then at least $k+1$ non-shadowed signal events happen before e in every execution consistent with the trace.

Algorithm 4: Initially $\hat{\tau}(e) = \tau'(e)$ for all events $e \in H$.

Repeat the following procedure until no more changes are possible.

Pick an event e.

If e is a wait event using semaphore S, let

- $W(S)$ be the set of wait events on semaphore S,
- k be the number of wait events $e_w \in W(S)$ such that $e_w \neq e$ and if $e_w \in H_i$ then $\hat{\tau}(e_w)[i] \leq \hat{\tau}(e)[i]$, and
- $R(e) = \{\hat{e} : \hat{e}$ is a signal event on S, $e \not\prec \hat{e}$ as indicated by the $\hat{\tau}$ time vectors, and $\hat{e}$ is not shadowed with respect to $e\}$

and $v_s =$ the $k+1$st component-wise minimum of $\hat{\tau}(\hat{e})$ for $\hat{e} \in R(e)$.

If e is not a wait event, let v_s be the 0 vector.

$$\hat{\tau}(e) = \overline{\max}(\hat{\tau}(e^p), \tau^{\#}(e), v_s)$$

End Algorithm 4.

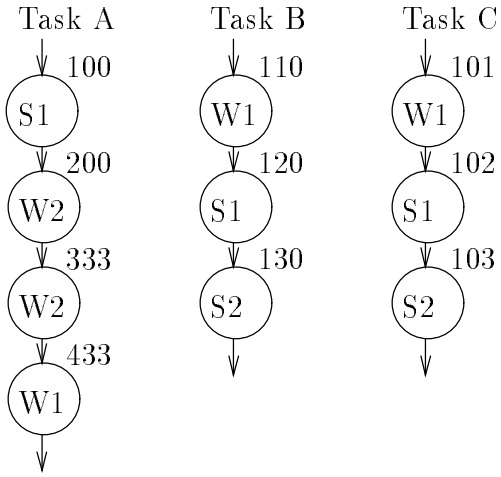

Figure 3.4: Expanding the Safe Order Relation

Figure 3.4 shows the new $\hat{\tau}$ time vectors generated when Algorithm 4 is executed starting with Figure 3.1.

Theorem 3: *Algorithm 4 generates only safe order relations, i.e., for any two distinct events e and $e' \in H$:*

$$\hat{\tau}(e) < \hat{\tau}(e') \Rightarrow e \prec e'$$

3.4 Running Time Analysis

The problem of calculating all safe order relations has been shown to be intractable [NM90]. We have presented a series of polynomial time algorithms that find many of the safe orderings that must occur in all executions that are consistent with the trace. Here we bound the execution times of these algorithms. We use m to represent the total number of events in the trace and n for the number of tasks executing events. Note that the sum of each time vector's components is bounded by m.

Assertion 1: *The time required by the initialization algorithm (Algorithm 2) is* $O(nm)$.

Every event is assigned an initial time vector value only once. By keeping an array containing the last time vector assigned for each task and a queue of signal events for each semaphore the relevant previous time vectors can be located in constant time. Order n steps suffice to compute the component-wise maximum. Therefore, the running time of Algorithm 2 is $O(nm)$.

Assertion 2: *The time required by the rewinding algorithm (Algorithm 3) is* $O(nm^3)$.

Since each iteration of Algorithm 3 decreases at least one component of a time vector, there can be at most m^2 iterations through the m events in the trace. By keeping the component-wise min of the time vectors for signals on for each semaphore, each event can be processed in $O(n)$ time (counting an update to the component-wise min after processing signal events). Therefore, the running time of Algorithm 3 is $O(nm^3)$.

Assertion 3: *The time required by the expanding algorithm (Algorithm 4) is* $O(nm^4)$.

As above, there are at most m^2 iterations. The time for processing signal events is dominated by the cost of processing wait events. Finding the set $R(e)$ for wait events is made easier by storing a pointer to the shadowing wait event (if any) with each signal event. Now the set $R(e)$, as well as the value k, can be found with a single pass through the trace, taking $O(n)$ time per event for time vector comparisons. The $k+1$st component-wise minimum can be calculated in $O(nm)$ time using a bucket sort on each component. Therefore, the overall time required by Algorithm 4 is in $O(nm^4)$.

In the above analysis of Algorithms 3 and 4, we used a very pessimistic m^2 bounded on the number of iterations required. In practice, we expect these algorithms will require only $O(m)$ iterations.

3.5 Adjusting the Time Vectors to Determine Concurrency

The previous algorithms compute a partial order that represents a safe order relation between the events from some trace H. Given any two events $e_i \in H_i$ and $e_j \in H_j$, if $\hat{\tau}(e_i) < \hat{\tau}(e_j)$ or $\hat{\tau}(e_j) < \hat{\tau}(e_i)$ then the two events are ordered. Otherwise, e_i and e_j are two unordered events. The unordered events need not necessarily be concurrent events. They may be constrained to occur sequentially, but in either order. In this case, we call them *unordered sequential* events. For example, if the program has a properly implemented lock around a critical region, then different executions may have tasks entering the critical region in different orders. In no execution, however, do two tasks concurrently enter the critical region.

To increase our understanding of a parallel program, we would like to distinguish those pairs of events that are concurrent in some consistent execution from the pairs of events which can happen in either order, but not concurrently. Unfortunately, the concurrent relation cannot be determined immediately from the time vectors. We cannot necessarily say e_i can happen concurrently with event e_j even if we know $\hat{\tau}(e_i) \parallel \hat{\tau}(e_j)$. As an example, in Figure 3.4, even though $\hat{\tau}(BW1) \parallel \hat{\tau}(CW1)$, the two W1 events cannot occur at the same time. But event e_i may happen concurrently with e_j only if $\hat{\tau}(e_i) \parallel \hat{\tau}(e_j)$. Determining whether or not two unordered events happen concurrently in some consistent execution is an NP-complete problem.

Next, we present an algorithm which detects critical regions and determines the associated unordered sequential event pairs[9]. The algorithm first determines if a pair of wait events on the same semaphore starts a pair of critical regions. If so, the algorithm then finds those unordered sequential event pairs within the critical regions by considering the effect of different execution orders of the two wait events.

The algorithm calculates two sets. The set Conc contains concurrent event pairs, while the set Seq contains unordered sequential event pairs. The event pairs in neither Conc nor Seq are ordered. Initially, we assume that all unordered events are potential concurrent events. As critical regions are detected, the algorithm moves the appropriate unordered sequential event pairs from Conc to Seq.

Algorithm 5: Initially let Conc $= \{\{e, e'\} : \hat{\tau}(e) \parallel \hat{\tau}(e')\}$ and Seq $= \emptyset$.

Repeat the following procedure until no more changes are possible.

Pick any two unordered wait events e and e' for semaphore S where $(e, e') \in$ Conc.

Let $G(e, e')$ be the set of wait events for semaphore S which precede either event e or e' (based on current time vectors $\hat{\tau}$).

Let $R(e, e') = \{e'' : e''$ is a signal event using S and e'' precedes e or $e'\} \cup \{e'' : e''$ does not follow either e or e' and e'' is not shadowed with respect to either e or e' $\}$.

Let $s = \mid R(e, e') \mid$ and $w = \mid G(e, e') \mid$.

- If $s - w \geq 2 \Longrightarrow e \parallel e'$, i.e., if there are enough signals for both waits to precede, then the two waits can happen concurrently.
- If $s - w = 1 \Longrightarrow \neg(e \parallel e')$, i.e., there is only one signal for a wait to precede, then we can conclude that they cannot happen concurrently. The starting points of critical regions have been found. The following procedure is used to determine unordered sequential event pairs in critical region.
  1. First, assume that event e happened before e'. Thus $w + 1$ wait events on S happened before e'. Use Algorithm 4 with $k = w + 1$ to calculate a new time vector for event e'. Continue with Algorithm 4 (with the modification that whenever the time vector for e' is calculated, event e is counted when determining k) to obtain a set of temporary time vectors.

[9]The algorithm may not, however, detect all of the unordered sequential event pairs. This is due primarily to the difficulty in detecting all of the safe orderings.

Let Seq_1 be the set of event pairs which are in Conc but are ordered by the temporary time vectors. After obtaining Seq_1, the original time vectors are restored. We can not yet move these events from Conc to Seq since they may be concurrent in executions where e' happens before e.

2. Now assume that event e' happened before e. Thus e is the $w+2$nd wait for S. As before, starting from the original time vectors, we run a modified Algorithm 4 (with the adjustment when the time vector for e is calculated). Let Seq_2 be the set of event pairs which are in Conc and are ordered by the resulting time vectors. Again, the original time vectors are restored after determining Seq_2.
3. The intersection of Seq_1 and Seq_2 gives the unordered event pairs in the critical regions. We therefore set $\text{Seq} = \text{Seq} \cup (\text{Seq}_1 \cap \text{Seq}_2)$ and $\text{Conc} = \text{Conc} - (\text{Seq}_1 \cap \text{Seq}_2)$.

- $s - w \leq 0$ means neither wait event can precede. In this case, there is a deadlock.

End Algorithm 5.

As an example, consider the two unordered wait events BW1 and CW1 in Figure 3.4. The two wait events cannot happen concurrently because there is only one signal (AS1) available for one of them to proceed in every consistent execution. They form two critical regions. In the executions where BW1 occurred before CW1, CW1 becomes the second wait on semaphore $S1$. Using Algorithm 4, we get time vectors as shown in Figure 3.5(a) where the event pairs {(BW1,CW1), (BW1,CS1), (BW1,CS2), (BS1,CW1), (BS1,CS1), (BS1,CS2)} appear ordered. Similarly, in the executions where CW1 occurred before BW1, event pairs {(BW1,CW1), (BS1,CW1), (BS2,CW1), (BW1,CS1), (BS1,CS1), (BS2,CS1)} are ordered as shown in Figure 3.5(b). At this point, we can conclude that the intersection of these two sets contains event pairs that are not concurrent in any executions, whenever BW1 happened before CW1 or CW1 before BW1. Therefore, {(BW1,CW1), (BW1,CS1), (BS1,CS1), (BS1,CW1)} are unordered sequential event pairs in the critical region, and can be moved from Conc to Seq.

4 Generalizing the Semaphore Model

The previous section described algorithms for systematically determining order relationships between events in a counting semaphore model. Here we generalize those results to the event-based synchronization mechanism provided by IBM Parallel FORTRAN [IBM88] and describe a working tool based on these algorithms. Although the algorithms presented in this section have the same initialize-rewind-expand top-level structure, modifications are needed to handle the IBM Parallel Fortran synchronization primitives.

An "event" in IBM Parallel Fortran is a particular programming construct, as opposed to its common usage in reference to any significant program step. Whenever it is not clear from the surrounding context, we will use IBM-event to refer to what IBM Parallel Fortran calls an "event", and trace record to refer to any significant program step.

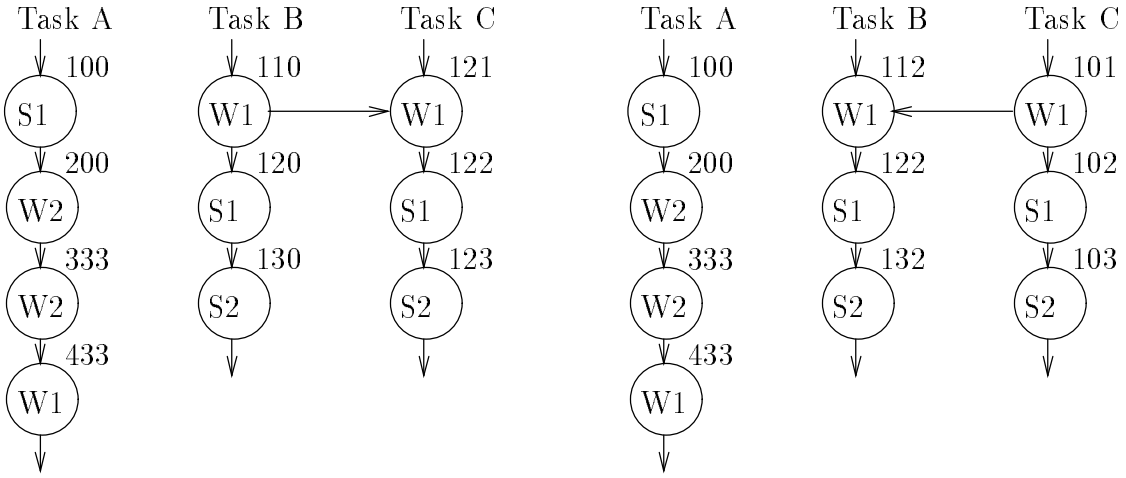

Figure 3.5: Detect Critical Regions

The remainder of this section is in four parts. Section 4.1 describes the IBM-events and their associated operations. Section 4.2 presents an algorithm to compute a safe order relation for a trace containing operations on IBM-events. Section 4.3 presents an algorithm that enhances the safe order relation with additional edges. Finally, Section 4.4 presents some sub-algorithms used by the algorithm in Section 4.3.

4.1 Parallel Events in IBM Parallel Fortran

In this section, we describe the parallel event facilities provided by IBM Parallel Fortran. Several subroutines are provided for the management of IBM-events. These subroutines are called to post events, wait for events, create and delete events, and initialize event parameters.

By using a common IBM-event, different tasks may synchronize their execution with each other. One or more tasks can signal the occurrence of an IBM-event to other tasks by calling PEPOST. Other tasks can synchronize their execution with the signaling task(s) by waiting for the event to be posted.

Every IBM-event has an *eventid, postcount, waitcount*, and *eventtype* associated with it. *Eventid* is an integer identifier for the IBM-event. *Postcount* specifies the number of times the event must be posted, and *waitcount* specifies the number of times the event must be waited on in order to complete a *work cycle* of the IBM-event. *Eventtype* can have a value of either 0 or 1. If the *eventtype* is 0, then the same task can post and/or wait on the event multiple times in the same cycle. A value of 1 indicates that the posting and waiting tasks within the same cycle must be unique. If a task posts a second time during the same work cycle, it will be suspended until the next cycle, where the second post can be counted. Similarly, if a task waits on the event for a second time in the same cycle, it will be suspended until the next cycle if the *eventtype* is 1.

The various combinations of these parameters provide very flexible synchronization patterns between tasks. As an example, an event with *postcount* =

3, *waitcount* = 1, and *eventtype* = 1 requires three unique tasks to signal their completion of a piece of work, before a single waiting task can continue.

An IBM-event can be viewed as a wait request queue, a post request queue, and a request processor. When a task makes a post or wait request, the task is suspended and placed in the appropriate queue until its request can be processed. The task continues execution after its request is processed.

The request processor has three processing phases: post request processing (PRP), wait request processing (WRP), and completed request processing (CP), as shown in Figure 4.1. When the request processor is in the PRP phase, requests are processed from the post request queue. When a number of post requests equal to the *postcount* have been processed, the processor advances to either WRP (when *waitcount* > 0) or CP (when *waitcount* = 0) phase.

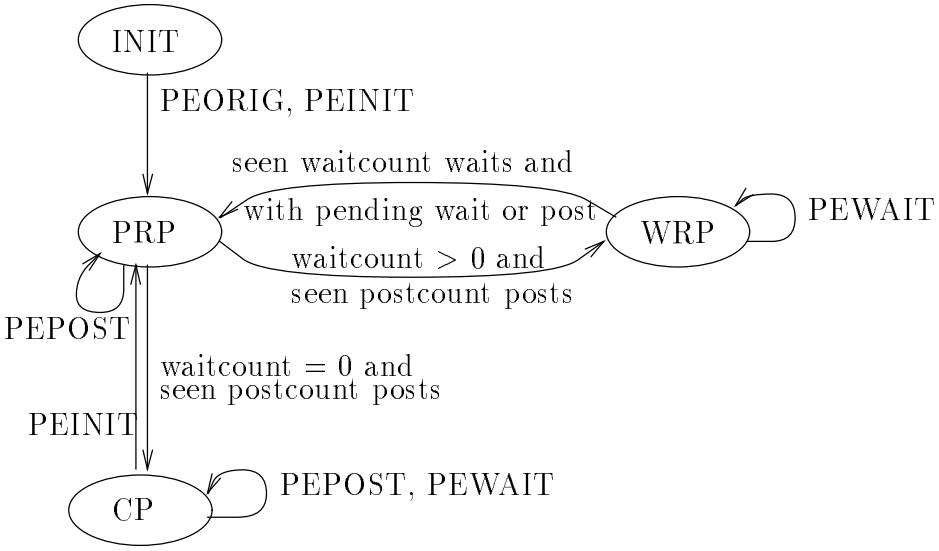

Figure 4.1: IBM Parallel Event

When the request processor is in the WRP phase, requests are processed from the wait request queue. When sufficient wait requests have been processed to satisfy the *waitcount*, the cycle is completed. If there are pending wait or post requests, the request processor advances to the PRP phase, automatically starting a new cycle.

When the request processor is in the CP state, requests from both the post and wait request queues are processed. Any tasks making subsequent wait or post requests will not be suspended. A new cycle can be started by calling PEINIT.

When an IBM-event is created, the *postcount*, *waitcount*, and *eventtype* are all set to 1. These values may be changed, either between cycles or before starting the first cycle, by a call to PEINIT. We view a call to subroutine PEINIT as creating a different IBM-event. When the first post or wait request is issued for the new event, the request processor enters the PRP phase.

4.2 Getting Safe Order Relations for IBM Events

For IBM-events, posts enable the waits in the same cycle, and waits enable the posts in the next cycle. In order to construct the canonical EHG for a trace containing IBM-events, we assume that the first *postcount* posts, from the trace H, correspond to the first cycle and the next *postcount* the second cycle, etc. Similarly for waits. These assumptions are used to quickly compute an initial set of time vectors, and the choice of canonical EHG does not affect the end results of our analysis.

The initial time vectors (from Algorithm 6) are computed such that the partial order represented by the time vectors is precisely the partial order represented by the EHG. For each event $e \in H_i$, the initial time vector value, $\tau(e)$, is simply the component-wise maximum of the time vectors for all of its immediate predecessors plus one in the ith position (i.e., the position of the local clock for task T_i).

Algorithm 6: *For all events $e \in H$, let*

$$\tau(e) = \overline{\max}(\tau(e_1^p), \ldots, \tau(e_k^p), \tau^{\#}(e))$$

where $e_1^p, \ldots, e_k^p$ are all the immediate predecessors of e in the EHG.
End Algorithm 6.

The time vector τ correctly represents the EHG. Event e has an earlier time vector than e' (i.e., $\tau(e) < \tau(e')$) if and only if e happened before e' in the execution specified by the EHG (i.e. $e \rightarrow e'$). Similarly, two events are unordered in the EHG if and only if their time vectors are unordered. Notice that the correspondence between post and wait events in the initial EHG represents only one of the executions which are consistent with the trace. It is not a safe order relation since for some events e and e', e' may happen before e or may be concurrent with e in a different execution even if e happened before e' in the consistent execution represented by the EHG. The next step (given by Algorithm 7) is to rewind the time vectors making the conservative assumption that any set of *postcount* posts could be responsible for unblocking any wait on the same IBM-event. This will result in a safe partial order. At this point we cannot tell if the edges between post and wait events are safe, so they are deleted from the canonical EHG[10], and the result is called the safe synchronization graph. For all remaining algorithms, the "predecessors" of a trace event refers to the predecessors in the safe synchronization graph (which is constant throughout the analysis).

Algorithm 7: *Repeat the following procedure until no further changes are possible.*

Pick an $e \in H$.

Let time vector v be the component-wise maximum of $\tau^{\#}(e)$ together with the time vectors for all of the immediate predecessors of e.

If e is not a wait event, then $\tau(e) = v$.

[10] In an actual EHG for IBM Parallel Fortran there will be edges in the EHG resulting from events other than post/wait, which are not deleted (see Section 5).

Otherwise, e is a wait on some event. Let $e_{p1} \ldots e_{pk}$ be all the posts on the same event and

$$\tau(e) = \overline{\max}(v, \overline{\min}_{postcount}(\tau(e_{p1}), \ldots, \tau(e_{pk})))$$

End Algorithm 7.

Algorithm 7 generates a partial order which is safe, i.e., if event e has an earlier time vector than e' (i.e. $\tau(e) < \tau(e')$), then e must happen before e' in all executions that are consistent with the given trace. Unfortunately, this safe order relation may be overly conservative. Some of the safe ordering edges may have been lost during the rewinding procedure. The next algorithm will add more safe ordering edges to the partial order.

4.3 Expanding the Safe Order Relations

The next step is to modify the assumption of the previous section that *any postcount* posts can trigger any wait on the same event. This will allow us to find more safe ordering edges between events in the partial order. The key to strengthening this assumption is reliable information as to which cycle(s) each post and wait can belong.

Definition 15: *For any post or wait $e \in H$, the* safe cycle number *for e is the earliest IBM-event cycle containing e in a consistent EHG.*[11] *A* cycle bound *on e is any lower bound on the safe cycle number for e.*

In particular, if a post has a safe cycle number or cycle bound n, then at least $(n-1) * postcount$ posts and $(n-1) * waitcount$ waits precede that post in every consistent execution. (Note that the particular events preceding the post may vary in different consistent executions.) Similarly, if a wait has a cycle bound n, then at least $n * postcount$ posts and $(n-1) * waitcount$ waits must precede that wait in every consistent execution.

Algorithm 7 assumed that any combination of the posts and waits can happen in the first cycle. This is why we used $\overline{\min}_{postcount}(\ldots)$ in Algorithm 7. If we can determine a cycle bound, $e.cycle$, for some wait event e, then we know that at least $postcount * e.cycle$ posts must precede event e. Therefore we use $\overline{\min}_{postcount * e.cycle}(\ldots)$ in Algorithm 8. In addition, if $e.cycle$ is greater than 1, then wait event e cannot be used to satisfy the $waitcount$ in cycle 1.

We next present the modified time vector algorithm (Algorithm 8) which finds additional safe orderings between events. This algorithm assumes that cycle bounds $e.cycle$ have been computed for each post and wait event in the trace. The following subsection contains a procedure for calculating these $e.cycle$ bounds.

Definition 16: *For an event $e \in H$, if e is a post or wait on some event, then $e.cycle$ is an estimated safe cycle number of e.*

[11] If an IBM-event has $waitcount = 0$ then all posts and waits on that IBM-event are considered to be in the first cycle.

Algorithm 8: *Repeat the following procedure until no more changes are possible.*

Pick an $e \in H$.

Let time vector v *be the component-wise maximum of* $\tau^{\#}(e)$ *and the time vectors for all of the immediate predecessors of* e *(in the safe synchronization graph).*

If e *is neither a post nor a wait event, then* $\tau(e) = v$.

Otherwise, e *is a post or wait on some event* S.

1. *Simple case where waitcount* $= 0$ *for IBM-event* S*: After postcount posts, all other posts and waits on* S *are non-blocking, and the cycle of the IBM-event never completes.*

   If e *is a post on some event* S*, then*

   $$\tau(e) = v$$

   Otherwise, e *is a wait on event* S*, then*

   $$\tau(e) = \overline{\max}(v, \overline{\min}_{postcount}(\tau(e_{p1}), \ldots, \tau(e_{pl}))$$

   where $e_{p1}, \ldots, e_{pl}$ *are all the posts on* S *with the following restrictions:*
   - *The posts following any wait on* S *(based on the current time vectors) are not included. They are "shadowed".*
   - *If* $eventtype = 1$ *then only the first post in each task is considered. Subsequent posts in the same task do not contribute towards fulfilling the postcount.*

2. *General case where waitcount* $\neq 0$ *for IBM-event* S*:*

   Calculate $e.cycle$*, the earliest cycle in which* e *can occur using Algorithm 10.*

   If e *is a post on event* S*, then* $\tau(e)$ *gets*

   $$\overline{\max}(v, \overline{\min}_{(e.cycle-1)*waitcount}(\tau(e_{w1}), \ldots, \tau(e_{wm}))$$

   where $e_{w1}, \ldots, e_{wm}$ *are the waits on* S *satisfying* $\tau(e) \not< \tau(e_{wi})$*, and* $e_{wi}.cycle < e.cycle$.

   Otherwise, e *is a wait on event* S*, and* $\tau(e)$ *gets*

   $$\overline{\max}(v, \overline{\min}_{e.cycle*postcount}(\tau(e_{p1}), \ldots, \tau(e_{pl})))$$

   where $e_{p1}, \ldots, e_{pl}$ *are the post events on* S *satisfying* $\tau(e) \not< \tau(e_{pi})$ *and* $e_{pi}.cycle \leq e.cycle$.

End Algorithm 8.

4.4 Safe Cycles

In order to calculate good cycle bounds for posts and waits, we need the following definition.

Definition 17: *For any trace record e which is a post or wait on some IBM-event S, let*

- *$W(e)$ be the set of waits on S preceding e based on the current time vectors, i.e., event e_w is in $W(e)$ if e_w is a wait event on S and $\tau(e_w) < \tau(e)$, and*
- *$P(e)$ be the set of posts on S preceding e based on the current time vectors, i.e., event e_p is in $P(e)$ if e_p is a post event on S and $\tau(e_p) < \tau(e)$.*

If the current time vectors represent a safe partial order, then all of the events in both $W(e)$ and $P(e)$ must happen before event e in every EHG consistent with the trace. A cycle bound for event e can be computed based on the following facts:

- If e is a wait on S, then $e.cycle \geq max(e'.cycle)$ for all $e' \in W(e)$ and $e' \in P(e)$. This is implied by the fact that e cannot happen before e' in any execution, i.e., it cannot appear in any earlier cycle.
- Similarly, if e is a post on S, then $e.cycle \geq max(e'.cycle)$ for all $e' \in P(e)$, and $e.cycle \geq max(e'.cycle) + 1$ for all $e' \in W(e)$. This is because the post e cannot occur until all posts and waits in previous cycles have completed.

Furthermore, since our algorithm calculates safe order relations which hold for all executions whose EHGs are consistent with the given trace, at some point we may find that the number of posts with a particular cycle number is greater than *postcount*, or the number of waits with the same cycle number is greater than *waitcount* (e.g. the rewinding procedure assumed that all posts and waits may happen in cycle 1). It may be difficult to figure out which post(s) or wait(s) must happen in any given cycle, however, in any cycle there are at most *postcount* posts from $P(e)$ and *waitcount* waits from $W(e)$. This can be used to increase cycle bounds, resulting in more safe orderings among events.

As an example, suppose that an IBM-event S has been initialized to need three posts and two waits in order to finish one cycle (i.e., $postcount = 3$ and $waitcount = 2$). Let e be a wait on S, $P(e)$ and $W(e)$ be the sets calculated according to Definition 17. Suppose in the set $P(e)$ there are four posts having cycle bounds of 1, one with a cycle bound 2, three with 3, five with 4, and four posts with cycle bounds equal to 5 (see Figure 4.2). Therefore, it is safe to conclude that the earliest cycle in which e may occur is the 5th cycle. However, we will see that this conclusion is overly conservative. In every consistent execution, at least one of the four posts which have cycle number 5 happens in a later cycle, since the *postcount* is equal to 3. The extra post(s) can be propagated to higher cycle(s). This means that e cannot happen in the 5th cycle and must happen later, because e must happen after all posts in $P(e)$.

Algorithm 9 can be used to determine a lower bound on the number of cycles required for a set of posts or waits. Given a set $W(e)$ or $P(e)$, let m be the maximum cycle bound for trace records in the set, let $c[i]$ be the number of trace records in

Cycle(i)	# of events in cycle i	justified number
1	4	3
2	1	2
3	3	3
4	5	3
5	4	6

Figure 4.2: Post and Wait Propagation

the set having cycle bound i, for $1 \leq i \leq m$. and let *count* represent *waitcount* if we are considering $W(e)$ or *postcount* otherwise. For the previous example, given the set of posts which happen before e, $m = 5$, the array $c = [4, 1, 3, 5, 4]$, and $count = postcount = 3$. After calling the *propagate* algorithm, the adjusted count numbers are shown in Figure 4.2.

Algorithm 9: *propagate*$(L, count)$

input: *L – set of trace records*
($W(e)$ or $P(e)$ where e is a post or wait on S);
count – # of records needed to satisfy a cycle.
output: *m – the max cycle bound of records in L;*
c[m] – the number of records which cannot
happen earlier than cycle m.
procedure:

Let m be the maximum cycle bound of trace records in set L, let c[i] be the number of trace records in the set with safe cycle number i, for $1 \leq i \leq m$.

```
for i := 1 to m - 1 begin
   if (c[i] > count) then begin
      c[i+1] = c[i+1] + c[i] - count;
      c[i] = count;
   endif;
endfor;
```

End Algorithm 9.

For the previous example, we can now conclude that the wait e cannot happen before the 6th cycle. The reason is that there are six previous posts on S which cannot happen in any cycle earlier than the 5th cycle, and at least three of them must happen in the 6th cycle or later since the *postcount* is set to 3. Because e must happen after all these posts, the earliest cycle in which e could happen is the 6th cycle. Notice that if e is a post on S then the earliest cycle of e will be the 7th cycle.

Based on these observations, we present the following algorithm which calculates a bound on earliest cycle in which a post or wait event e could occur. The algorithm does the propagation and then computes the new cycle bound based on $c[m]$, the number of events propagated to the last cycle. The key to the algorithm is that the extra events (including those propagated to the last cycle) must be spilled into later cycle(s), pushing up the safe cycle number for event e. Algorithm 10 is used in the time vector computation given by algorithm 8.

Algorithm 10: *safe cycle number estimation.*

input: *a trace record e which is either a post or wait on some event S.*
output: *$e.cycle$, the new cycle bound for e.*
procedure: *Let $W(e)$ and $P(e)$ be the sets of previous waits and posts respectively (Definition 17). Based on these two sets $W(e)$ and $P(e)$, a wait cycle count c_w and a post cycle count c_p are calculated.*

1. *If e is a wait on S, first calculate c_w by calling propagate$(W(e), waitcount)$ and let*

$$c_w = m + \lfloor c[m]/waitcount \rfloor;$$

then call propagate$(P(e), postcount)$ and let

$$c_p = m + \lfloor (c[m] - 1)/postcount \rfloor.$$

2. *Otherwise, e is a post on S, first calculate c_w by calling propagate$(W(e)$, $waitcount)$ and let*

$$c_w = m + 1 + \lfloor (c[m] - 1)/waitcount \rfloor;$$

then calculate c_p by calling propagate$(P(e), postcount)$
and let

$$c_p = m + \lfloor c[m]/postcount \rfloor;$$

Let $e.cycle = max(c_w, c_p)$.

When $eventtype = 1$, a task cannot post twice or wait twice in the same cycle. In this case, we have the following adjustment:

- *If e is a wait and $e \in H_i$, then*

$$e.cycle = max(e.cycle, e'.cycle + 1)$$

where $e' \in W(e)$ and $e' \in H_i$.

- *Otherwise e is a post and $e \in H_i$, then*

$$e.cycle = max(e.cycle, e'.cycle + 1)$$

where $e' \in P(e)$ and $e' \in H_i$.

End Algorithm 10.

5 A Prototype for IBM Parallel Fortran

In IBM Parallel FORTRAN, a task can be explicitly created, assigned work, and waited for until the work assigned to it has been completed. A task can also be implicitly created by parallel loops and parallel cases. Tasks can be executed concurrently. A *parallel lock* can be used to prevent interference between tasks during manipulation of critical data areas. A *parallel event* permits explicitly created tasks to synchronize their execution through intertask signaling.

The IBM Parallel FORTRAN Trace Facility can automatically record important events during the execution of a parallel program, providing useful information about the execution. The Trace Facility produces a series of time-stamped trace records during execution of a parallel program. At least one trace record is generated for each of the following operations:

- Start and end of program execution,
- Origination and termination of tasks,
- Assignment and completion of task work,
- Waiting for tasks to complete work,
- Start and end of parallel loop and parallel case execution, and
- Use of parallel locks and parallel events.

In addition to the time stamp, each trace record identifies the kind of action, the program unit and task performing the action, the virtual FORTRAN processor used, and the actual CPU on which the program unit was executing. Additional information specific to the kind of action may also be recorded.

To build an EHG from an IBM trace we need to determine the enabling-blocking pairings for all synchronization events.

1. For dispatching and scheduling (enabling) events, the corresponding task begin is the blocking event.
2. For IBM-events, posts enable the waits in the same cycle, and waits enable the posts in the next cycle.
3. For task completion (enabling) events, the corresponding event waiting for the task completion is the blocking event.

The algorithms described in the previous section have been implemented in a trace analyzer, and several traces have been analyzed. The trace analyzer can construct the event history graph and calculate the time vector values for all events in the trace. The final time vectors represent a safe partial order among events. By comparing the time vectors, we can distinguish many ordered events from unordered events. Combining this with variable reference information from START [McD89], the trace analyzer reports those data races which can happen in any executions whose EHG is consistent with the given trace. The trace analyzer is implemented mainly in C++, and part of the code is implemented in C.

A graphical tool, built on top of the X Window System, has also been implemented to assist programmers in comprehending the trace information recorded during program execution and generated using the above algorithms. The tool allows the user

to browse the safe partial order computed and display the detected races. Using a pointing device the user may request that various information related to a selected node be displayed. The types of information displayed include:

- all information known about the event from the trace,
- highlight all events that must happen before the selected event,
- highlight all events that must happen after the selected event,
- highlight all events that may happen concurrently the selected event, and
- the program source with the line generating the event highlighted.

6 Related Work

Recently, much research has been directed towards determining the partial ordering of events in parallel and distributed systems. Previous models have assumed point-to-point communication which makes it very easy to determine which events were caused by which other events (e.g. "message received by B from A" is clearly caused by "message sent by A to B"). Unfortunately the synchronization models supported by several parallel programming languages allow for anonymous communication, where the partner is unknown. Examples of anonymous communication include locks, semaphores, and monitors.

Emrath, Ghosh, and Padua [EGP89] present a method for detecting non-determinacy in parallel programs that utilize fork/join and event style synchronization instructions with the `Post, Wait`, and `Clear` primitives. They construct a *Task Graph* from the given synchronization instructions and the sequential components of the program that is intended to show the guaranteed orderings between events. For each `Wait` event node, all `Post` nodes that might have triggered that `Wait` are identified. An edge is then added from the closest common ancestor of these `Post` events to the `Wait` event node. Although their algorithm is simple, it may be computationally complex. Rather than repeatedly computing the common ancestor information, we use time vectors to calculate the guaranteed execution order.

Netzer and Miller [NM89] present a formal model of a parallel program execution. Their model includes fork/join parallelism and synchronization using semaphores. They distinguish between an *actual data race*, which is a data race exhibited by the particular program execution generating the trace, and a *feasible data race*, which is a data race that could have been exhibited due to timing variations. Their approach and ours differ in the amount of trust placed in the trace. They rely on the trace for their ordering information. For example, when two tasks try to enter critical regions protected by a binary semaphore, their algorithm will say that the critical regions are ordered. Under their definitions there is neither an actual nor feasible data race even if two tasks write to a shared variable in the critical regions. We view the ordering relationships in the trace with suspicion, and wish to generate race reports in this situation[12].

[12] If the critical regions contain non-commutative operations, then the race to enter the regions can affect the remainder of the execution [Wan90].

Dinning and Schonberg [DS90] present a method of detecting access anomalies in parallel programs "on-the-fly". They use a mechanism, which is similar to time vectors, to identify concurrent operations in a program execution. Some compaction methods are used to reduce the storage needed for reader and writer sets. If a variable is involved in multiple data races, then some of those races may not be reported. However, at least one of the data races involving the variable will be reported by their algorithm. They need explicit coordination between tasks in order to construct the partial order execution graph (POEG). The POEG represents the order relations between operations for just one of many possible consistent executions.

We believe that it is more helpful to analyze sets of executions rather than just one specific execution based on some trace information. We feel that, in terms of detecting data races by trace analysis, it is critical to distinguish the *ordered* events from the *unordered*, potentially *concurrent*, events. In this paper we presented a collection of algorithms that extend previous work in computing partial orders. The algorithms presented compute a partial order containing only *must occur* type orderings from a linearly ordered trace containing anonymous synchronization. The algorithms presented in this paper make few assumptions about specific trace features and can be adjusted to work with traces generated by many parallel systems.

7 Summary

Debugging parallel programs is more difficult than debugging sequential programs. One of the fundamental problems encountered when debugging parallel programs is detecting unintended non-determinacy in parallel programs. Tools which automatically detect non-determinacy can be used to debug timing and synchronization errors when the program is expected to be determinate. The tools we are developing help one find the data races which can lead to non-determinacy. This paper presents a method for detecting data races, which is based on analyzing a program trace from an execution of a parallel program.

When debugging parallel programs, it is critical to find the order and concurrency relationships among operations in the program. One of the most difficult tasks in trace analysis is determining the timing relationships between the events performed by the parallel program. Although several parallel systems include facilities for creating a trace of the significant events, the sequential nature of the trace makes it difficult to determine which events could have happened in either order or in parallel. The problem is made even more difficult in the anonymous synchronization model, where there is no clear correspondence between the blocking and enabling events in the trace. The problem of calculating all safe order relations has been shown to be co-NP-hard by Netzer and Miller [NM90].

This paper contains a series of polynomial time algorithms for extracting useful information from sequential traces with anonymous synchronization. The first algorithm is very similar to the vector timestamp methods of Fidge and Mattern [Fid88, Mat88]. The other algorithms systematically manipulate these vectors of timestamps

in order to discover pairs of events that must be ordered in every execution which is consistent with the trace.

Some parallel programming environments view a parallel execution as a linear sequence of events. We feel that this is misleading – an execution is more properly viewed as a partial ordering on the events. Fidge and Mattern have pioneered the use of time vectors to represent these partial orders. We have extended this approach by using time vectors to analyze sets of executions rather than just capturing a single execution.

A working trace analyzer has been implemented, and some experiments have been performed. The current implementation analyzes traces generated by IBM Parallel Fortran and includes a graphical trace browser. The trace analyzer reports various data race conditions in parallel programs by finding unordered/concurrent events and variable access conflicts.

Acknowledgements

This work was supported by IBM under agreement SL 88096.

References

[AP87] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *Proc. International Conf. on Parallel Processing*, pages 721–727, 1987.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), September 1965.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1990.

[EGP89] P. A. Emrath, S. Ghosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '89*, November 1989. Reno, NV.

[EP88] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988.

[HMW90] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Analyzing traces with anonymous synchronization. In *Proc. International Conference on Parallel Processing*, August 1990.

[HMW91] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Detecting data races from sequential traces. In *Proc. of Hawaii International Conference on System Sciences*, pages 408–417, 1991.

[IBM88] *Parallel FORTRAN language and library reference.* IBM, 1988.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms*, 1988.

[MC88] B. P. Miller and J-D. Choi. Breakpoints and halting in distributed systems. In *Proc. Int. Conf. on Distributed Computing Systems*, June 1988.

[McD89] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, June, 1989.

[NM89] R. Netzer and B. P. Miller. *Detecting Data Races in Parallel Program Executions.* Technical Report 894, University of Wisconsin-Madison, November 1989.

[NM90] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel progra m executions. In *Proc. International Conf. on Parallel Processing*, pages 93–97, 1990.

[Tay84] R. N. Taylor. *Debugging Real-Time Software in a Host-Target Environment.* Technical Report, U.C. Irvine Tech. Rep. 212, 1984.

[Wan90] J-Z. Wang. *Debugging Parallel Programs by Trace Analysis.* Technical Report, Masters Thesis UCSC-CRL-90-11, 1990.