*TR 908* *January 19, 1990*

# On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions

*Robert H. B. Netzer*
*netzer@cs.wisc.edu*

*Barton P. Miller*
*bart@cs.wisc.edu*

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

## Abstract

This paper presents results on the complexity of computing event orderings for shared-memory parallel program executions. Given a program execution, we formally define the problem of computing orderings that the execution *must have* exhibited or *could have* exhibited, and prove that computing such orderings is an intractable problem.

We present a formal model of a shared-memory parallel program execution on a sequentially consistent processor, and discuss event orderings in terms of this model. Programs are considered that use fork/join and either counting semaphores or event style synchronization. We define a *feasible program execution* to be an execution of the program that performs the same events as an observed execution, but which may exhibit different orderings among those events. Any program execution exhibiting the same data dependences among the shared data as the observed execution is feasible. We define several relations that capture the orderings present in all (or some) of these feasible program executions. The *happened-before*, *concurrent-with*, and *ordered-with* relations are defined to show events that execute in a certain order, that execute concurrently, or that execute in either order but not concurrently. Each of these ordering relations is defined in two ways. In the *must-have* sense they show the orderings that are guaranteed to be present in all feasible program executions, and in the *could-have* sense they show the orderings that could potentially occur in at least one feasible program execution due to timing variations. We prove that computing any of the must-have ordering relations is a co-NP-hard problem and that computing any of the could-have ordering relations is an NP-hard problem.

## Key Words

Event ordering, nondeterminacy, parallel processing, race conditions, synchronization.

---

Research supported in part by National Science Foundation grant CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

Copyright © 1990 Robert H. B. Netzer, Barton P. Miller.

A condensed version of this paper appears in *Proc. of the 1990 Int'l Conference on Parallel Processing*, St Charles, IL, August 1990, pp. II-93–II-97.

## 1. Introduction

In an execution of a shared-memory parallel program, the order in which some events execute may not be enforced by (explicit or implicit) synchronization, but instead may occur by chance. Even if two events are explicitly synchronized to force a certain order during one execution, the same events may occur in a different order during another execution. Due to nondeterministic timing variations, the program may, on different occasions, execute exactly the same events but exhibit different orderings among those events. We formally define several relations that capture the orderings present in all (or some) such alternate executions, and prove that computing these relations is an intractable problem. We consider programs executing on sequentially consistent processors that use fork/join and either counting semaphores or event variables.

We present a formal model of a shared-memory parallel program execution, and define event orderings in terms of this model. Given a program execution, *P*, we characterize other program executions that execute exactly the same events as *P* but which may exhibit different event orderings. Any program execution exhibiting the same data dependences among the shared data as *P* will execute the same events as *P*. The set of all such alternate program executions, called *feasible program executions*, is defined by considering all the different orderings that could allow the data dependences exhibited by *P* to occur. The various orderings that must have been exhibited (or could have been exhibited) by all such feasible program executions are captured by defining several ordering relations: *happened-before*, *concurrent-with*, and *ordered-with*. Each of these relations is defined in the *must-have* sense and in the *could-have* sense. The must-have relations show orderings that are guaranteed to occur in all feasible program executions, while the could-have relations show orderings that could potentially occur due to timing variations. We prove that computing the must-have relations is a co-NP-hard problem, and that computing the could-have relations is an NP-hard problem. These results are shown to hold for programs that use fork/join and either counting semaphores or event style synchronization (using the Post, Wait, and Clear primitives). We also show that these results hold when computing the orderings occurring in *all* program executions exhibiting the same events as a given execution, regardless of whether the original shared-data dependences occur.

## 2. Program Execution Model

In this section, we briefly present a formal model of shared-memory parallel program executions. The model contains the objects that represent a program execution (such as which statements were executed and in what order), and axioms that characterize properties those objects can possess. In subsequent sections, we use the model to characterize behavior that an execution *might* have exhibited (such as alternate event orderings) and behavior that an execution *must* have exhibited (such as obeying the semantics of its synchronization operations).

Our model† provides a formalism for reasoning about shared-memory parallel program executions that does

† This section is a brief presentation of the model that was first presented by us in an earlier paper[10], and is based on Lamport's theory of concurrent systems[8].

not assume the existence of atomic operations. We consider the class of shared-memory parallel programs that execute on sequentially consistent processors[7] and that use fork/join and either counting semaphores or event style synchronization. A program execution is described by a collection of *events* and two relations over those events. Each event represents an execution instance of a set of (consecutively executed) program statements. We distinguish between two types of events: a *synchronization event* is an instance of some synchronization operation, and a *computation event* is an instance of a group of statements belonging to the same process, none of which are synchronization operations. The *temporal ordering* relation†, $\xrightarrow{T}$, describes the temporal ordering among events in the program execution; $a \xrightarrow{T} b$ means that $a$ completes before $b$ begins (in the sense that the last action of $a$ can affect the first action of $b$), and $a \overset{T}{\not\longrightarrow} b$ means that $a$ and $b$ execute concurrently (i.e., neither completes before the other begins). The *shared-data dependence* relation, $\xrightarrow{D}$, indicates when one event causally affects another; $a \xrightarrow{D} b$ means that $a$ accesses a shared variable that $b$ later accesses, where at least one of the accesses is a modification to the variable.‡

We define a *program execution*, $P$, to be a triple, $\langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, where $E$ is a finite set of events, and $\xrightarrow{T}$ and $\xrightarrow{D}$ are the relations over $E$ described above. The temporal ordering and shared-data dependence relations must satisfy several axioms that describe properties a valid program execution must possess[10]. We omit these axioms here as they are not required to prove our results.

## 3. Problem Statement

In a given program execution, the temporal ordering between some events is not always ‘‘guaranteed’’. Another execution of the program could perform exactly the same events, but due to nondeterministic timing variations, could exhibit a different temporal ordering among those events. In this section, we characterize program executions exhibiting such alternate temporal orderings, and define several relations that capture the orderings present in all (or some) of these program executions. In subsequent sections, we prove that computing these relations is an intractable problem.

### 3.1. Feasible Program Executions

Given a program execution, $P$, a *feasible program execution for P* (or just a *feasible program execution*, when $P$ is implied) describes an execution of the program that performs exactly the same events as $P$, but which may exhibit different temporal orderings. Any execution that exhibits the same shared-data dependences as $P$ will

---

† Throughout this paper we use superscripted arrows to denote relations, and write $a \rightarrow b$ as a shorthand for $\neg(a \rightarrow b)$, and $a \longleftrightarrow b$ as a shorthand for $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$.

‡ This definition of data dependence is different from the standard ones[3, 6] in that our definition combines the notions of flow-, anti-, and output-dependence, and does not explicitly state the variable involved.

execute exactly the same events as $P$[†] This result can be proven by showing that the execution result of each statement instance depends only upon the values of the variables it reads, and that the program's input and the shared-data dependences uniquely characterize these values for each step in the computation[9]. Therefore, a program execution $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$ is a feasible program execution for $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ if

(F1) $E' = E$, and

(F2) $P'$ satisfies the axioms of the model[10], and

(F3) $a \xrightarrow{D} b \Rightarrow a \xrightarrow{D'} b$.

These conditions state that any valid program execution (i.e., a program execution obeying the axioms mentioned in Section 2) possessing the same events and shared-data dependences as $P$ describes an execution that is guaranteed to have *potentially* occurred. This statement holds even if the program executes nondeterministic statements, since $P'$ is still capable of executing the same events as $P$. We denote the set of feasible program executions for $P$, as characterized above, by $F(P)$ (or just $F$, when $P$ is implied).

### 3.2. Ordering Relations

Given a program execution, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, and the set, $F$, of feasible program executions for $P$, we define several relations (shown in Table 1) that summarize the temporal orderings present in the feasible program executions in $F$.

| | Must-Have | Could-Have |
|---|---|---|
| Happened-Before | $a \xrightarrow{MHB} b \Leftrightarrow$ $\forall \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle \in F,\ a \xrightarrow{T} b$ | $a \xrightarrow{CHB} b \Leftrightarrow$ $\exists \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle \in F,\ a \xrightarrow{T} b$ |
| Concurrent-With | $a \xleftrightarrow{MCW} b \Leftrightarrow$ $\forall \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle \in F,\ a \xleftrightarrow{T} b$ | $a \xleftrightarrow{CCW} b \Leftrightarrow$ $\exists \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle \in F,\ a \xleftrightarrow{T} b$ |
| Ordered-With | $a \xleftrightarrow{MOW} b \Leftrightarrow$ $\forall \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle \in F,\ \neg(a \xleftrightarrow{T} b)$ | $a \xleftrightarrow{COW} b \Leftrightarrow$ $\exists \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle \in F,\ \neg(a \xleftrightarrow{T} b)$ |

**Table 1. Ordering Relations For A Set, *F*, Of Feasible Program Executions**

Each relation type (*happened-before*, *concurrent-with*, or *ordered-with*)[‡] is defined to capture both the *must-have* and *could-have* orderings. The must-have relations describe orderings that are guaranteed to be present in all feasible program executions in $F$, while the could-have relations describe orderings that could potentially occur in

† For this statement to hold, interactions with the external environment must be modeled as shared-data dependences.

‡ We use double arrows ( $\longleftrightarrow$ ) to denote the concurrent-with and ordered-with relations to emphasize that they are symmetric.

at least one of the feasible program executions in *F*. The happened-before relations show events that execute in a specific order, the concurrent-with relations show events that execute concurrently, and the ordered-with relations show events that execute in either order but not concurrently.

## 4. Related Work

The problem of computing event orderings has been previously addressed by several researchers[1, 2, 5]. Given a program execution, Emrath, Ghosh, and Padua[2], and Helmbold, McDowell, and Wang[5] attempt to compute orderings that are guaranteed to occur in any other execution exhibiting the same events, irrespective of the original shared-data dependences. Callahan and Subhlok[1] consider event orderings in the context of static analysis of parallel FORTRAN programs. Emrath, Ghosh, and Padua[2] attempt to compute a graph that shows the must-have and could-have orderings for programs that use event style synchronization. However, since they do not consider the ordering constraints imposed by the shared-data dependences, their method sometimes overlooks some orderings. Helmbold, McDowell, and Wang[5] consider programs that use counting semaphores, and present algorithms for computing only some of the must-have orderings. Callahan and Subhlok[1] prove that computing orderings that are guaranteed to occur in *all* executions of a given program is a co-NP-hard problem, and present a data-flow framework for computing some of these orderings.

Emrath, Ghosh, and Padua[2] describe a method for computing the ‘‘guaranteed run-time ordering’’ between events in program executions that use fork/join and event style synchronization (using Post, Wait, and Clear operations). They construct a graph (called a *task graph*) that contains a single node for each synchronization event in the program execution. *Task Start* and *Task End* edges are added to represent orderings imposed by fork and join operations, and *Machine* edges are added to represent orderings of events belonging to the same process. *Synchronization* edges are also added to represent guaranteed orderings imposed by synchronization. For each Wait node, all Post nodes that might have triggered that Wait are identified. A Post might trigger a Wait if there is no path from the Wait to the Post (which would indicate that the Wait must have preceded the Post), and no path from the Post to the Wait that includes a Clear node. Synchronization edges are then added from the closest common ancestors of these Posts to the Wait. The resulting graph is intended to show a guaranteed ordering between two events iff there is a path between the nodes representing those events. However, since their method does not account for the orderings imposed by the shared-data dependences, the graph sometimes shows no ordering when indeed an ordering is enforced by a shared-data dependence.

Consider the program fragment shown in Figure 1a. Its task graph, for the case when the first created task completely executes before the other two created tasks, is shown in Figure 1b. In this execution, there is a shared-data dependence from the statement instance ‘‘X := 1’’ to ‘‘**if** X=1 **then**’’. The dotted lines represent orderings imposed by the fork operation, and the solid line represents a guaranteed ordering (drawn from the closest common ancestor of the two Post nodes to the Wait node). In this graph, there is no path between the two Post nodes. However, the two Posts cannot execute in either order. Due to the shared-data dependence from the statement instance ‘‘X := 1’’ to ‘‘**if** X=1 **then**’’, it is not possible for the right-most Post to execute before the left-most

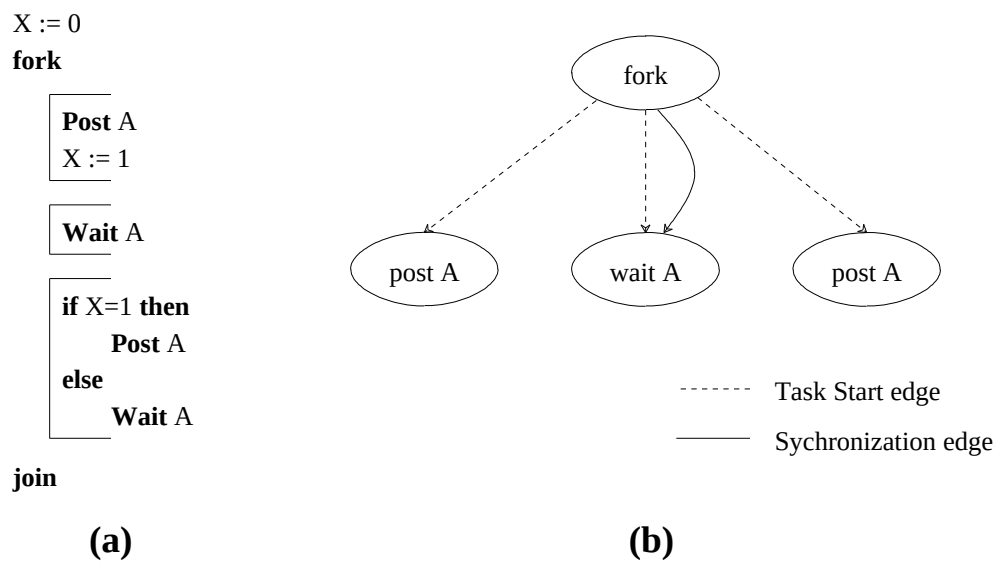


**Figure 1. Example Program Fragment and a Task Graph**

Post. If this shared-data dependence does not occur, the **else** clause will execute, causing a Wait to be issued instead of the right-most Post. This example illustrates that even if the programmer does not intentionally introduce synchronization with shared variables, some events are nevertheless ordered by the shared-data dependences. Any method that attempts to compute could-have orderings must therefore consider these dependences. We prove in the next section that exhaustively computing the orderings for program executions that use event style synchronization (with Clear operations) is an intractable problem.

Given a trace of a program that uses counting semaphores, Helmbold, McDowell, and Wang[5] present an algorithm to compute some of the must-have orderings by computing *safe orderings*. A ordering is safe if the orderings it contains are guaranteed to occur in all executions that exhibit the same events (regardless of the shared-data dependences). Their algorithm operates in three phases. First, for each semaphore, they order the $i^{th}$ *V* event before the $i^{th}$ *P* event in the trace. The transitive closure of the union of this ordering with the intra-process orderings defines their *happened before* relation. This relation is unsafe because another execution (performing the same events) might exhibit a different pairing among the semaphore operations. In the second phase, their happened before relation is altered so that each *V* event is ordered before all *P* events on the same semaphore. The resulting relation is safe, but overly conservative. To sharpen the relation, the third phase adds additional safe orderings by considering that only some *P* events can actually execute after certain *V* events. Their algorithms run in

polynomial time since they compute only some of the must-have-happened-before orderings. The resulting ordering relation is therefore a subset of our $\xrightarrow{\text{MHB}}$ relation. In the next section we prove that computing the entire $\xrightarrow{\text{MHB}}$ relation is a co-NP-hard problem.

Callahan and Subhlok[1] consider the problem of static analysis of FORTRAN programs without loops that use parallel DO and CASE and event style synchronization (without Clear operations). They prove that determining the orderings that are guaranteed to occur in *all* executions of such a program is a co-NP-hard problem. They also present a data-flow framework for computing some of these guaranteed orderings.

## 5. Complexity of Computing Ordering Relations

In this section, we describe the complexity of computing the ordering relations defined in Section 3. We prove that the problem of computing any of the must-have ordering relations (i.e., $\xrightarrow{\text{MHB}}$, $\xleftrightarrow{\text{MCW}}$, or $\xleftrightarrow{\text{MOW}}$) is co-NP-hard and that the problem of computing any of the could-have ordering relations (i.e., $\xrightarrow{\text{CHB}}$, $\xleftrightarrow{\text{CCW}}$, or $\xleftrightarrow{\text{COW}}$) is NP-hard. These results are shown to hold for programs that use counting semaphores, and for programs that use event style synchronization (using the Post, Wait, and Clear primitives). Finally, we briefly comment on intractability results for computing the ordering relations when the shared-data dependences are not considered.

### 5.1. Counting Semaphores

*Theorem 1*. Given a program execution, $P = \langle E, \xrightarrow{\text{T}}, \xrightarrow{\text{D}} \rangle$, that uses counting semaphores, the problem of deciding whether $a \xrightarrow{\text{MHB}} b$, $a \xleftrightarrow{\text{MCW}} b$, or $a \xleftrightarrow{\text{MOW}} b$ (i.e., any of the must-have ordering relations) is co-NP-hard.

*Proof.* We present a proof only for the must-have-happened-before relation ( $\xrightarrow{\text{MHB}}$ ); proofs for the other relations are analogous. We give a reduction† from 3CNFSAT[4] such that any Boolean formula is not satisfiable iff $a \xrightarrow{\text{MHB}} b$ for two events, $a$ and $b$, defined in the reduction. Let an arbitrary instance of 3CNFSAT be given by a set of $n$ variables, $V = \{X_1, X_2, \cdots, X_n\}$, and a Boolean formula $B$ consisting of $m$ clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause is a disjunction of three literals (a literal is any variable or its negation, from $V$). From such an instance of 3CNFSAT, we construct a program consisting of $3n+3m+2$ processes that uses $3n+m+1$ semaphores (all semaphores are assumed to be initialized to zero). The execution of this program simulates a nondeterministic evaluation of the Boolean formula $B$. Semaphores are used to represent the truth values of each variable and clause. As we will show, the execution exhibits certain orderings iff $B$ is not satisfiable.

For each variable, $X_i$, construct the following three processes:

† This reduction was motivated by the ones Taylor[11] constructed to prove that certain static analysis problems are NP-complete.

| | | |
|---|---|---|
| P($A_i$) | P($A_i$) | V($A_i$) |
| V($X_i$) | V($\overline{X}_i$) | P(*Pass* 2) |
| . | . | V($A_i$) |
| . | . | |
| . | . | |
| V($X_i$) | V($\overline{X}_i$) | |

where " $\cdots$ " indicates as many V($X_i$) (or V($\overline{X}_i$)) operations as occurrences of the literal $X_i$ (or $\overline{X}_i$) in the formula *B*. The semaphores $X_i$ and $\overline{X}_i$ are used to represent the truth value of variable $X_i$; a signaling of semaphore $X_i$ (or $\overline{X}_i$) represents the assignment of *True* (or *False*) to variable $X_i$. The above processes operate in two passes. The first pass is a nondeterministic guessing phase in which each variable used in the Boolean formula is assigned a unique truth value. This assignment is accomplished by allowing either the V($X_i$) operations or the V($\overline{X}_i$) operations to proceed, but not both. The second pass, which begins after semaphore *Pass* 2 is signaled, is used only to ensure that the program does not deadlock; the semaphore operations that were not allowed to execute during the first pass are allowed to proceed.

For each clause, $C_j$, construct the following three processes:

| | | |
|---|---|---|
| P($L_1$) | P($L_2$) | P($L_3$) |
| V($C_j$) | V($C_j$) | V($C_j$) |

where $L_1$, $L_2$, and $L_3$ are the semaphores corresponding to the literals in clause $C_j$. The semaphore $C_j$ represents the truth value of clause $C_j$. This semaphore will be signaled if the truth assignments guessed during the first pass cause clause $C_j$ to evaluate to *True*.

Finally, create the following two processes:

| | | | |
|---|---|---|---|
| | | *a*: | **skip** |
| | P($C_1$) | | V(*Pass* 2) |
| | $\cdots$ | | $\cdots$ |
| | P($C_m$) | | V(*Pass* 2) |
| *b*: | **skip** | | |

where there are *n* V(*Pass* 2) operations (one for each variable). Event *b* is reached only after semaphore $C_j$, for each clause *j*, has been signaled.

Since the program contains no conditional statements or shared variables, every execution of the program executes the same events and exhibits the same shared-data dependences (i.e., none). For any execution, we claim that *a* $\xrightarrow{\text{MHB}}$ *b* iff *B* is not satisfiable.

To show the ''if'' part, assume that $B$ is not satisfiable. Then there is always some clause, $C_j$, that is not satisfied by the truth values guessed during the first pass. Therefore, no V($C_j$) operation is issued during the first pass. Event $b$ cannot execute until this V operation is issued, which can then only be done during the second pass. The second pass does not occur until after event $a$ executes, so event $a$ must precede event $b$.

To show the ''only if'' part, assume that $a \xrightarrow{\text{MHB}} b$. That is, there is no execution in which $b$ either precedes $a$ or executes concurrently with $a$. For a contradiction, assume that $B$ is satisfiable. Then some truth assignment can be guessed during the first pass that satisfies all of the clauses. Event $b$ can then execute before event $a$, contradicting the assumption. Therefore, $B$ cannot be satisfiable.

Since $a \xrightarrow{\text{MHB}} b$ iff $B$ is not satisfiable, the problem of deciding $a \xrightarrow{\text{MHB}} b$ is co-NP-hard. By similar reductions, programs can be constructed such that the non-satisfiability of $B$ can be determined from the $\xleftrightarrow{\text{MCW}}$ or $\xleftrightarrow{\text{MOW}}$ relations. The problem of deciding these relations is therefore also co-NP-hard. □

*Theorem 2.* Given a program execution, $P = \langle E, \xrightarrow{\text{T}}, \xrightarrow{\text{D}} \rangle$, that uses counting semaphores, the problem of deciding whether $a \xrightarrow{\text{CHB}} b$, $a \xleftrightarrow{\text{CCW}} b$, or $a \xleftrightarrow{\text{COW}} b$ (i.e., any of the could-have ordering relations) is NP-hard.

*Proof.* The reduction used in the proof of Theorem 1 can be used to prove that deciding $\xrightarrow{\text{CHB}}$ is an NP-hard problem. Using the events defined in the above reduction, we can show that $b \xrightarrow{\text{CHB}} a$ iff $B$ is satisfiable, showing that deciding $\xrightarrow{\text{CHB}}$ is NP-hard. As above, similar reductions can be constructed to prove that deciding the $\xleftrightarrow{\text{CCW}}$ and $\xleftrightarrow{\text{COW}}$ relations is also NP-hard. □

The above proofs do not make use of the general counting ability of counting semaphores, and therefore also hold for programs that use binary semaphores. In addition, the above results can be shown to hold for a program execution that uses a single counting semaphore by a reduction from the problem of sequencing to minimize maximum cumulative cost[4].

### 5.2. Event Style Synchronization

*Theorem 3.* Given a program execution, $P = \langle E, \xrightarrow{\text{T}}, \xrightarrow{\text{D}} \rangle$, that uses event style synchronization (with Post, Wait, and Clear primitives), the problem of deciding whether $a \xrightarrow{\text{MHB}} b$, $a \xleftrightarrow{\text{MCW}} b$, or $a \xleftrightarrow{\text{MOW}} b$ (i.e., any of the must-have ordering relations) is co-NP-hard.

*Proof.* The proof is similar to the proof of Theorem 1 (for programs that use semaphores), and hinges on the ability to implement two-process mutual exclusion using only synchronization operations (i.e., no shared variables). We give a reduction from 3CNFSAT that produces a program operating in the same manner as described in Theorem 1. For each variable, $X_i$, construct the following process:

Post($A_i$)
Post($B_i$)
**fork**
  Clear($A_i$)
  Wait($B_i$)
  Post($X_i$)

  Clear($B_i$)
  Wait($A_i$)
  Post($\bar{X}_i$)
**join**

Although these processes can deadlock, when they do not exactly one of Post($X_i$) or Post($\bar{X}_i$) will be issued.

For each clause, $C_j$, construct the following three processes:

| | | |
|---|---|---|
| Wait($L_1$) | Wait($L_2$) | Wait($L_3$) |
| Post($C_j$) | Post($C_j$) | Post($C_j$) |

where $L_1$, $L_2$, and $L_3$ are the event variables corresponding to the literals in clause $j$.

Finally, create the following two processes:

| | | | |
|---|---|---|---|
| | | $a$: | **skip** |
| | Wait($C_1$) | | Post($A_1$) |
| | $\cdots$ | | Post($B_1$) |
| | Wait($C_m$) | | $\cdots$ |
| $b$: | **skip** | | Post($A_n$) |
| | | | Post($B_n$) |

These processes operate in a manner analogous to those created using semaphores, described in Theorem 1. We claim that $a \xrightarrow{\text{MHB}} b$ iff the Boolean formula is not satisfiable. The proof is analogous to the argument given in Theorem 1. □

*Theorem 4.* Given a program execution, $P = \langle E, \xrightarrow{\text{T}}, \xrightarrow{\text{D}} \rangle$, that uses event style synchronization, the problem of deciding whether $a \xrightarrow{\text{CHB}} b$, $a \xleftrightarrow{\text{CCW}} b$, or $a \xleftrightarrow{\text{COW}} b$ (i.e., any of the could-have ordering relations) is NP-hard.

*Proof.* The proof is analogous to the proof of Theorem 2 (for programs that use semaphores). The required reductions are similar to the reduction given in Theorem 3 above. □

The above proofs rely on the implementation of two-process mutual exclusion with only synchronization primitives (i.e., no shared variables). Theorems 3 and 4 made use of the Clear primitive to implement two-process

mutual exclusion. We are currently investigating the complexity of computing the ordering relations for program executions that do not use the Clear primitive.

### 5.3. Ordering Relations Ignoring Shared-Data Dependences

The related work outlined in Section 4 addresses the event ordering problem by using a different notion of feasibility than we presented here. These methods attempt to compute some (or all) of the orderings exhibited by *all* program executions exhibiting the same events as a given program execution, regardless of whether the original shared-data dependences occur. The complexity results given in this section extend directly to this case. Since the programs constructed by the various reductions contain no shared-data dependences, the proofs suffice to show that even when the original shared-data dependences are ignored and *all* alternate program executions are considered, computing the ordering relations is still an intractable problem. As we showed, these results hold for programs that use counting semaphores or event style synchronization.

## 6. Conclusion

This paper has presented results showing that, given a program execution, computing event orderings that the execution might have exhibited (or must have exhibited) is an intractable problem. Given a program execution, *P*, we defined a *feasible program execution* to be an execution of the program that performs the same events as *P* but which may exhibit different temporal orderings among those events. Any program execution that possess the same shared-data dependences as *P* is a feasible program execution. To capture the orderings exhibited by all (or some) of these program executions, we defined several ordering relations. The *happened-before*, *concurrent-with*, and *ordered-with* relations were defined to show events that execute in a certain order, that execute concurrently, or that execute in either order but not concurrently. Each of these ordering relations was defined in two ways. In the *must-have* sense they show the orderings that are guaranteed to be present in all feasible program executions, and in the *could-have* sense they show the orderings that could potentially occur at least one feasible program execution due to timing variations.

We proved that computing any of the must-have ordering relations is a co-NP-hard problem and that computing any of the could-have ordering relations is an NP-hard problem. These results were shown to hold for programs that use counting semaphores and for programs that use event style synchronization (using the Post, Wait, and Clear primitives). In addition, these results also hold for programs that use only a single counting semaphore or multiple binary semaphores. It is currently an open problem whether these results hold for program executions that use only a single binary semaphore or event style synchronization without Clear operations. We also showed that these results hold when computing the orderings occurring in *all* program executions exhibiting the same events as a given execution, regardless of whether the original shared-data dependences occur. An implication of these results is that exhaustively detecting all data races potentially exhibited by a given program execution[10] is an intractable problem.

**References**

[1] Callahan, David and Jaspal Subhlok, ‘‘Static Analysis of Low-Level Synchronization,’’ *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[2] Emrath, Perry A., Sanjoy Ghosh, and David A. Padua, ‘‘Event Synchronization Analysis for Debugging Parallel Programs,’’ *Supercomputing ’89*, pp. 580-588 Reno, NV, (November 1989).

[3] Ferrante, J., Karl J. Ottenstein, and Joe D. Warren, ‘‘The Program Dependence Graph and its Use in Optimization,’’ *ACM Trans. on Programming Languages and Systems* **9**(3) pp. 319-349 (1987).

[4] Garey, Michael R. and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co. (1979).

[5] Helmbold, David P., Charles E. McDowell, and Jian-Zhong Wang, ‘‘Analyzing Traces with Anonymous Synchronization,’’ *Proc. of the 1990 Intl. Conf. on Parallel Processing*, pp. 70-77 St. Charles, IL, (August 1990).

[6] Kuck, D. J., R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, ‘‘Dependence Graphs and Compiler Optimizations,’’ *Conf. Record of the Eighth ACM Symp. on Principles of Programming Languages*, pp. 207-218 Williamsburg, VA, (January 1981).

[7] Lamport, Leslie, ‘‘How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,’’ *IEEE Trans. on Computers* **C-28**(9) pp. 690-691 (September 1979).

[8] Lamport, Leslie, ‘‘The Mutual Exclusion Problem: Part I — A Theory of Interprocess Communication,’’ *Journal of the ACM* **33**(2) pp. 313-326 (April 1986).

[9] Mellor-Crummey, John M., ‘‘Debugging and Analysis of Large-Scale Parallel Programs,’’ *Ph.D. Thesis, also available as Computer Science Dept. Tech. Rep. 312*, Univ. of Rochester, (September 1989).

[10] Netzer, Robert H. B. and Barton P. Miller, ‘‘Detecting Data Races in Parallel Program Executions,’’ in *Languages and Compilers for Parallel Computing*, ed. D. Gelernter, T. Gross, A. Nicolau, and D. Padua, MIT Press (1991). Also appears in *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

[11] Taylor, Richard N., ‘‘Complexity of Analyzing the Synchronization Structure of Concurrent Programs,’’ *Acta Informatica* **19** pp. 57-84 (1983).