Analysis of Event Synchronization in A Parallel Programming Tool

David Callahan* Ken Kennedy† Jaspal Subhlok†

Abstract

Understanding synchronization is important for a parallel programming tool that uses dependence analysis as the basis for advising programmers on the correctness of parallel constructs. This paper discusses static analysis methods that can be applied to parallel programs with event variable synchronization. The objective is to be able to predict potential data races in a parallel program. The focus is on how dependences and synchronization statements inside loops can be used to analyze complete programs with parallel loop and parallel case style parallelism.

1 Introduction

Parallel programming is an intellectually demanding task. One of the most difficult challenges in the development of parallel programs for asynchronous shared-memory systems is avoiding errors caused by inadvertent data sharing, often referred to as *data races*, which can lead to unpredictable results. These races correspond to *data dependences* [AK87, Wol82], where a data dependence links a pair of accesses to the same variable or memory location, with one such access being a write. At Rice, we have developed a system called Ptool [BBC+88, Hen87] that permits a programmer to review the potential data races, if a particular loop were to execute in parallel. Ptool has been in use at several locations and the experience gained by using it was the principal motivation for this research.

*Tera Computer Company, 400 North 34th Street, Suite 300, Seattle, WA 98103

†Department of Computer Science, Rice University, Houston, Texas 77251-1892

This research was supported by the National Science Foundation under grants ASC-8518578 and CCR-8809615 and by the IBM Corporation.

When we first implemented Ptool, we assumed that programmers would use it on sequential Fortran programs that were to be converted to a parallel form suitable for execution on a shared memory multiprocessor. Hence, it only handled sequential programs. In practice, however, its users preferred to write a program in parallel form, invoking Ptool only when a data race was discovered during testing. Users of Ptool felt that a tool of this nature should accept "parallel Fortran" as input and understand its features, including synchronization statements. Also, in many instances Ptool would report a data race that the user with superior knowledge of his program knew did not exist. This made it difficult for users to locate the actual sources of problems in their programs. An effort was made to overcome these limitations in a redesign of Ptool and in the design of ParaScope, a parallel programming environment being developed at Rice.

There are two main reasons why Ptool would sometimes report a false data race. Since data dependence analysis assumes a dependence between two references to the same variable whenever absence of dependence cannot be proved, many dependences assumed to cause a data race did not actually exist. An effort was made to make the computation of parallelization-inhibiting dependences as accurate as possible. In particular, since dependences on variables that are private to a loop body do not inhibit parallelization, accurate automatic detection of private variables significantly reduces the number of false data races that are reported. The other reason for false reports of data races was Ptool's inability to understand synchronization. Since user inserted synchronization can be used to eliminate potential non-determinism caused by dependent statements specified to execute in parallel, it is important to analyze synchronization in a program and infer its effect on program behavior. Callahan and Kennedy [CK87] have developed a method for dealing with critical region style synchronization. In this paper we discuss the handling of event variable style synchronization. Extensions to other synchronization mechanisms are briefly discussed

```
DOALL 10 I = 1,10
    :
END DOALL
```

(a)

```
PARALLEL BEGIN
    S1
PARALLEL
    :
PARALLEL
    SN
PARALLEL END
```

(b)

```
POST(ev)
  :
WAIT(ev)
```

(c)

Figure 1: Asynchronous parallel constructs and Synchronization statements

in the conclusions.

Through the proper use of event synchronization, the user can insure that a particular dependence is always satisfied by forcing the code at the sink of the dependence to wait until the code at the source has been executed. This paper develops a method for determining which dependences in a program cannot possibly result in a data race because of schedule restrictions enforced by event posts and waits.

Unlike dynamic methods that use some form of runtime tracing [AP87, MC88, Sch89], our method is based entirely on static analysis and does not use any runtime information. Most previous work on static analysis of parallel programs has focused on task parallelism [Tay83, AM85, CS88, BK89] and is not suitable for the analysis of parallel loops. The method we propose overcomes this limitation and generalizes the loop analysis methods in [EP88].

This paper has 8 sections. Section 2 defines the computation paradigm to which our methods apply and Section 3 defines the problem to be solved in the paper. In section 4 we present a program graph structure, which is the framework for our analysis. Sections 5 and 6 discuss two independent methods of analyzing event variable synchronization. Section 7 is a discussion of handling of `CLEAR` statement. Section 8 contains conclusions.

2 A Parallel Language

Our methods are designed for parallel programs written for shared memory multiprocessors. The programming language assumed is Fortran 77, with parallel constructs that are supported on most parallel Fortran implementations. We now state the extensions to Fortran used for expressing parallelism and synchronization.

Parallel DO and Parallel Case

Parallelism can be expressed via two language constructs: parallel DO and parallel case. These are reasonably standard [FJS85, Par88] and we make no unusual assumptions. The parallel DO is syntactically and semantically very similar to the Fortran 77 DO loop. Here we use `DOALL` to indicate a parallel DO (illustrated in Figure 1(a)). When control reaches a parallel DO, all instances (iterations) of the loop body are started and proceed concurrently, but asynchronously. A parallel DO completes and control continues to the next program statement only when all instances of the loop body have completed.

The parallel case describes a fixed set of tasks to be executed concurrently. An example is illustrated in Figure 1(b), where each `Si` represents a list of statements. When control flow reaches a parallel case, each `Si` begins execution concurrently, but asynchronously, with the others. A parallel case completes and control continues to the next program statement only when all of the `Si` have completed.

Parallel constructs may be nested inside parallel constructs. Branches are not allowed from within a parallel construct to outside the parallel construct or vice versa.

We use the term **task** to refer generically to an instance of the body of a parallel DO or one of the statement lists of a parallel case. We will use the term **thread** to refer to a task while it is executing. When a parallel construct is executed, one thread is created for every task in the construct.

Synchronization

Many different mechanisms are used for synchronization in programs written for shared memory multiprocessors. These include locks, semaphores, event variables and many others. However, since this paper develops analysis methods for event variable synchronization, we assume that all synchronization is provided via **event** variables. An event variable is always in one of two states: *clear* or *posted*. The initial value of an event variable is always *clear*. The value of an event variable can be set to *posted* with the `POST` statement shown in Figure 1(c). The value of an event variable can be "tested" with a `WAIT` statement, also shown in Figure 1(c). A `WAIT` statement suspends execution of the thread which executes it until the specified event variable's value is set to *posted*. A `CLEAR` statement resets the value of an event variable to *clear*. For the methods developed in this paper, we assume there are no `CLEAR` statements. In section 7 we discuss the implications of the presence of `CLEAR` statements.

3 Correct Parallel Execution

In this section we define "correctness" for parallel execution and informally define the analysis problem to be solved in the paper.

Sequential Execution

A canonical sequential execution schedule of a parallel program can be defined by describing a canonical sequential execution schedule for each of the parallel constructs. The sequential execution of a parallel DO is simply the obvious sequential iteration of the instances of the body, as if it were a standard Fortran DO loop. The sequential execution of a parallel case is the sequential execution of each independent statement list, in the textual order in which they occur in the program.

If during sequential execution of a parallel program, a `WAIT` statement is executed and the corresponding event variable is not already posted, then we say that the sequential execution of that program is **undefined**. If a program's sequential execution is defined for all input data sets, we classify that program as **serializable**. We assume serializability throughout this paper, but the techniques developed can be used for non-serializable programs too. Our primary interest is in debugging scientific programs (where parallelism is primarily a performance issue) using event variable synchronization for software pipelining, so the restriction to serializable programs is reasonable.

Correctness and Dependence Analysis

A **data dependence** [AK87, Kuc78] exists between two statements if they both access some memory location M and at least one of them modifies that location:

Definition 1 *A data dependence $s_1 \Delta s_2$ (read "s_2 depends on s_1") exists if there is a memory location M such that both s_1 and s_2 access M, at least one of them stores into M, and, in the sequential execution of the program, s_1 is executed before s_2. A data dependence is* carried *by a loop L if s_1 and s_2 are statement instances in different iterations of L.*

Much research has focused on identifying data dependence relationships and using them for loop restructuring transformations [KKP+81]. Data dependences capture the constraints on a program's data flow that are necessary to ensure correct results if a program is executed in parallel. If there are no data dependences between a pair of statements, the results of executing them are independent of the order in which they are executed.

An incorrect parallel program is defined to be a program whose output may differ from the sequential execution output on some input data:

Definition 2 *A parallel program is* **correct** *relative to sequential execution (schedule-independent) if all parallel executions compute the same results as the sequential execution.*

Since the execution state of a program consists of the program counter for each thread and the value for each memory cell, a difference between a parallel execution and the sequential execution must be recorded in memory at some point. In particular, for some memory location an incorrect program must contain a read/write, write/read, or write/write/read sequence that holds in the sequential execution but does not hold in some parallel execution. This notion is captured in the next definition:

Definition 3 *A data dependence $s_1 \Delta s_2$ is* **preserved** *if, for all parallel executions, s_2 begins execution after s_1 has completed execution, whenever both are executed.*

In a language with asynchronous parallel constructs, the semantics of a program are well-defined (schedule-independent) if there are no dependences between concurrent tasks. Thus, a parallel program is correct (relative to sequential execution) if all dependences are preserved on all parallel executions.

When a parallel program is not provably correct, any dependence which is not preserved is a potential error in the parallel program. Our objective is to be able to isolate such dependences and present them to the user as parallel execution hazards.

4 A Framework to Analyze Synchronization

In this section we describe a framework that forms the basis of our analysis of synchronization. The objective is to have a program representation that exposes the effects of synchronization and control flow, as well as data dependences, particularly inside loops.

Dependence and Synchronization Distance Vectors

We associate an **iteration number** with every execution instance of a loop body, which is the order of execution of that instance of the loop body in the program's canonical sequential execution. Thus every execution of a statement is associated with a unique **iteration vector** whose components are the iteration numbers of all the enclosing loop bodies.

Data dependence analysis associates distance vectors with data dependences, denoting the number of iterations across which a dependence is carried. Here we define dependence distance vectors in terms of iteration vectors. Let there be a data dependence D from statement s_1 to s_2 (s_2 depends on s_1) due to an access to a variable in shared memory. If s_1 executing with iteration vector $\vec{i_1}$ always acceses the same memory location as s_2 executing with iteration vector $\vec{i_2}$, then $\vec{\delta_D} = \vec{i_2}$ - $\vec{i_1}$ is the **dependence distance vector** of data dependence D.

```
DOALL I
    DOALL J
        WAIT(EV(I-1,J-1))
S       C(I,J) = A(I,J)
      D A(I+2,J+2) = B(I,J)
        POST(EV(I,J))
    END DOALL
END DOALL
```

Figure 2: D is a distance (2,2) dependence edge and S is a distance (1,1) synchronization edge

In our discussion we assume that all data dependence vectors are integer vectors that can be computed by data dependence analysis techniques. While this is by far the most common case, many exceptions are found in scientific programs. Our analysis procedures can use partial dependence distance information if constant integer vectors do not exist or are not computable. In particular, if only direction vectors are available (it is only known whether the distance terms are positive, negative or zero) they can be used for partial analysis.

A distance vector term is meaningful for a `POST` and `WAIT` statement pair acting on the same event variable. Let s_1 be a `POST` statement and s_2 be a `WAIT` statement accessing a common set of memory locations (they act on the same, possibly array, event variable). If s_1 executing with iteration vector $\vec{i_1}$ always posts the same memory location that s_2 executing with iteration vector $\vec{i_2}$ waits on, then $\vec{\delta_S} = \vec{i_2}$ - $\vec{i_1}$ is the **synchronization distance vector** for the *synchronization edge* from s_1 to s_2. Figure 2 shows an example of a dependence and a synchronization distance vector.

The synchronization distance vector defines the relative distance in terms of iteration numbers for which a pair of event variable directives are meaningful. Standard data dependence analysis techniques are normally able to compute the synchronization distance vectors, which are almost always integer vectors. We mark the synchronization distance components that are not simple integers by "$*$". They are not of use in our methods directly but can have special significance. One such special case is discussed later in the section on initialization edges.

The Synchronized Control Flow Graph

Analysis of synchronization is similar to analysis of control flow in that no resolution is needed in straight-line code without synchronization statements. The primary program representation used in this section is a modified form of the control flow graph, in which the nodes represent a form of basic blocks and edges represent both control flow and "synchronization flow".

The **control flow subgraph** is a directed graph, $G_c = (N, E_c)$, where the nodes N are program blocks [ASU86] built under the condition that a block in N must contain no control flow* and may contain at most one `WAIT`, which must be the first statement in the block, and at most one `POST`, which must be the last statement in the block. An edge represents a possible transfer of control from one block to another. Both parallel case and parallel DO are represented as a pair of nodes. One node represents the point where new threads are created (the fork node) and the other represents the point where threads are destroyed (the join node). Each task in a parallel construct is required to be a single-entry single-exit region. For each task inside the parallel construct, there is an edge from the fork node to the entry of the task and an edge from the exit of the task to the exit node. There is no "back edge" from the bottom of a parallel DO to the top.

All edges in this graph are labeled with a distance vector which represents the iteration span over which the edge is meaningful. The control flow edges are trivially labeled with all zero distance vectors implying that the execution of the sink block with a particular iteration vector must follow the execution of the source block with the same iteration vector, if both are executed. However, in the case of the control flow edge from the last block in a sequential loop to the top of the loop, the last block in the previous iteration of the loop body must finish execution before the first block in the next iteration can begin execution. Correspondingly, the distance vector component corresponding to that loop is one, while all others are zero. A control flow edge from block b_i to b_j with distance vector $\vec{\delta}$ is denoted by $\langle b_i, b_j, \vec{\delta}\rangle$.

*This restriction is stronger than is necessary. Internal control flow can be allowed, for example, loops and complete if-then-else blocks without synchronization statements in their bodies can be a part of a block.

Synchronization Edge

Control Flow Edge

```
POST(EV(1))
DOALL I = 1, 100
   C(I) = C(I) + 3
   WAIT(EV(I))
   T = T + C(I)
   POST(EV(I+1))
END DOALL
```

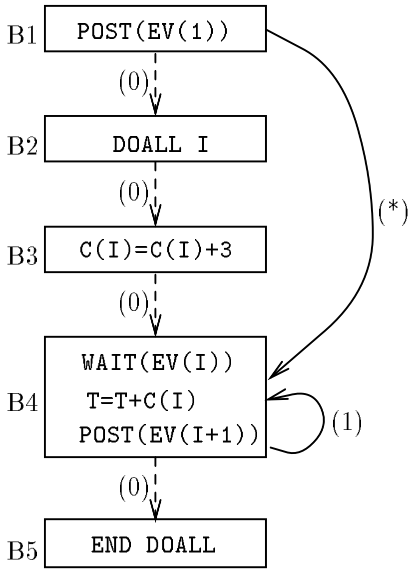

Figure 3: Synchronized Control Flow Graph for a Simple Loop

The **synchronization flow subgraph** is a directed graph, $G_s = (N, E_s)$, where the nodes N are the same as in the control flow subgraph but the edges, E_s, link blocks which post events to blocks which wait on the same events. In particular, $\langle b_1, b_2, \vec{\delta}\rangle \in E_s$ if the last statement in block b_1 is `POST(`f`(ev))` and the first statement in block b_2 is `WAIT(`g`(ev))` where f`(ev)` and g`(ev)` are subscript expressions referencing the same event variable `ev` and $\vec{\delta}$ is the synchronization distance vector, whose value is determined by the arguments to `POST` and `WAIT` statements as discussed earlier.

The **synchronized control flow graph**, SCG, is the combination of the control flow subgraph and synchronization flow subgraph: $SCG = (N, E)$, where E is the direct union $E_c \cup E_s$.

Our objective is to analyze the synchronized control flow graph SCG and conclude which data dependences are preserved by the synchronization and control flow constraints, and hence cannot lead to data races. In the next two sections we present two different methods to solve this problem, both based on the SCG.

5 A Data Flow Formulation

We present our first method for identifying preserved dependences, which are the dependences protected by event variable synchronization. This method is based on setting up the problem as a system of data flow equations over sets of pairs containing a program block and a distance vector term.

Preserved Sets

We shall use the SCG described earlier to compute the information from which we can deduce if a given dependence is preserved. For this purpose we introduce the notion of preserved sets.

Intuitively, the preserved set for a block (say b_i) is the set of blocks that are guaranteed to execute before b_i, if they do execute in a program run. This concept of preserved sets is sufficient for programs without loops, which execute each program block at most once. However, in a program with loops, each block may be executed several times. That is, a block can have many execution instances. It is clearly undesirable to compute preserved sets for each possible execution instance of a block, and to have execution instances of blocks as members. Fortunately, the way synchronization and control flow constraints are specified, (factors that cause preserved sets to exist) it is usually possible to store this information compactly. Since the synchronization edges are relative between loop iterations, the final preserved sets can also be specified in this manner. We shall introduce a predicate on the execution of blocks and then define preserved sets:

$\mathit{ExecutedBefore}((i, \vec{l}), (j, \vec{m})) \iff$ Block b_i with iteration vector $\vec{l}$ will complete execution before block b_j with iteration vector $\vec{m}$ begins execution, if both are executed.

Definition 4 *The* **Preserved** *sets for each block b_i are defined by :*
$\mathit{Preserved}(j) =$

$$\{(i, \vec{\delta}) \mid \forall \vec{m}(\mathit{ExecutedBefore}((i, \vec{m} - \vec{\delta})(j, \vec{m}))\}$$

$$SCPreserved(i) = CPreserved(i) \cup SPreserved(i) \tag{1}$$

$$CPreserved(i) = \begin{cases} \bigcup_{\langle j,i,\vec{\delta}\rangle \in E_c} \left(Addedge(j,i,\vec{\delta})\right) & \text{if } b_i \text{ is a parallel case``join'' node;} \\ \bigcap_{\langle j,i,\vec{\delta}\rangle \in E_c} \left(Addedge(j,i,\vec{\delta})\right) & \text{otherwise ;} \end{cases} \tag{2}$$

$$SPreserved(i) = \bigcap_{\langle j,i,\vec{\delta}\rangle \in E_s} \left(Addedge(j,i,\vec{\delta})\right) \tag{3}$$

$$Addedge(j,i,\vec{\delta}) = \left(\{(k, \vec{\delta}+\vec{\delta'}) \mid ((k,\vec{\delta'}) \in SCPreserved(j))\} \cup \{(j,\vec{\delta})\}\right)^{\dagger} \tag{4}$$

Figure 4: Equations to compute SCPreserved Sets

Thus $(i,\vec{\delta}) \in Preserved(j)$ implies that a distance $\vec{\delta}$ dependence from block b_i to b_j is preserved. Clearly once the preserved sets are computed, it is a simple matter to verify which dependences are preserved.

We have proved in another paper [CS88] that computation of preserved sets is NP-hard, even for simple programs without loops.

Data Flow Equations

Since the computation of preserved sets is NP-hard, we describe a method to compute a conservative approximation of preserved sets, which we call *SCPreserved* sets. For a block b_i, every member of SCPreserved(i) must belong to Preserved(i) but not vice versa.

The SCPreserved sets are computed as the least fixed point of a set of dataflow equations over the *SCG*. These equations define the SCPreserved set for a block as a function of its control flow and synchronization flow predecessors. The information collected at each node about which blocks must precede it in execution is transmitted to the successor blocks in each iterative computation with the dataflow equations. The fixed point of these equations provide the final SCPreserved set for each block. We state these equations in Figure 4 and informally justify their correctness.

In each application of these equations to a block, a new SCPreserved set for the block is computed. These equations are repeatedly applied to each block until a fixed point is reached.

The basic propagation of values along edges is done by the Addedge function defined in equation (4). We illustrate the application of this function in Figure 5.

Initially
SCPreserved(x) $= \{(\text{a},\vec{\delta_1}), (\text{b}, \vec{\delta_2})\}$
SCPreserved(y) $= \phi$

Addedge(x,y, $\vec{\delta}$) $= \{(\text{a},\vec{\delta_1}+\vec{\delta}), (\text{b},\vec{\delta_2}+\vec{\delta}), (\text{x},\vec{\delta})\}$

If the only edge into 'y' is from 'x' we get
SCPreserved(y)$= \{(\text{a},\vec{\delta_1}+\vec{\delta}), (\text{b},\vec{\delta_2}+\vec{\delta}), (\text{x},\vec{\delta})\}$

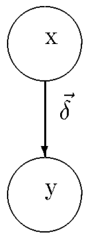

Figure 5: Information Propagation function *Addedge*

The SPreserved set of a block b_i, computed in equation (3), contains all the entries (block-distance vector pairs) that are determined to be in the preserved set of b_i as a result of information obtained from the synchronization flow predecessors of b_i and distance vectors of synchronization edges incident on b_i. For an instance of a block (say $(b_i, \vec{m})$), which has at least one synchronization edge coming into it, to begin execution, the relevant iteration of one of its immediate predecessors in the synchronization flow subgraph must have finished execution. That is, there must exist a synchronization edge $\langle b_j, b_i, \vec{\delta}\rangle$ such that $(b_j, \vec{m}-\vec{\delta})$ has finished execution. It follows that the set of entries in the intersection of all synchronization flow predecessors, after the distance corresponding to the edge into b_i has been added in the distance term, can be included in the preserved set of b_i. The sets of terms for each predecessor is computed in equation (4) with Addedge function, and the intersection is taken in equation (3).

The CPreserved sets (in equation (2)) are computed in an analogous manner using information from the control flow predecessors and the the control flow edges. In case of the join node of a parallel case, *all* (not just one) control flow predecessors must have finished execution, so we take union instead of intersection.

The SCPreserved set contains all entries independently determined to be in the preserved set using control flow or synchronization flow information. This is stated in equation (1).

† if the new entry is inside the iteration space

Finally, when using equation (4), we check that the entries being added have relevance in the iteration space of loops enclosing the block.

In this system of equations, entries are only added (and never deleted) from preserved sets. If iteration bounds of all the loops in a program are known, this system of equations must reach a fixed point, since there are a finite number of entries that can be members of a SCPreserved set.

In the rest of this section we discuss some practical considerations which help us make this method more accurate and faster.

Initialization Edges

Certain facts can only be represented approximately in a system based on the *SCG*. In Figure 3, there are two synchronization edges incident on block B_4. The first one is an *initialization edge* which ensures that the first iteration of the loop does not wait on the `WAIT` statement in B_4 (and cause a deadlock). The second synchronization edge from B_4 to itself ensures that the kth iteration of B_4 completes execution before the $(k+1)th$ iteration begins. The fact that the two synchronization edges refer to non-intersecting sets of iterations cannot be inferred from the synchronized control flow graph. When computing the preserved set for B_4, the computing system assumes that for any iteration of B_4, B_1 could have posted the location in the event variable array on which B_4 waits to begin execution. Thus, it cannot infer that every iteration of B_4, except the first one, waits for the previous iteration to finish, and hence a distance (1) dependence from B_4 to itself is preserved.

In general, we can restrict our attention to synchronization and control flow edges between blocks at the same level of nesting, to decide whether a loop carried dependences between two blocks is preserved. Thus, in the above example, we would ignore the synchronization edge from B_1 to B_4, if our interest was in loop carried dependences inside the loop. Identifying and accounting for initialization edges is essential for obtaining accurate results for preserved set computation.

Synchronization Cycles

The process of reaching a fixed point in solving the system of equations may be slow because of the effect of certain loops in the *SCG* which lead to a large number[‡] of *cyclic* entries such as $(p, 3^*)$, with the meaning that the block p with all distances that are a multiple of 3, belong to a preserved set. Detection of such behavior due to such "synchronization cycles" leads to more efficient solutions. We shall illustrate this informally and then give the related definitions.

[‡]of the order of the size of loop iteration space

Consider the program in Figure 3. The backward synchronization edge from the block B4 to itself ensures that all iterations of B4 with iteration number less then k must complete execution before the kth iteration can start execution. This implies that $(B4, j)$ for all j less than the iteration range of the loop (100 here) belong to the preserved set for $B4$. Here $B4$ constitutes a synchronization cycle and the set of entries mentioned above can be written as $(B4, 1^*)$ implying that all $(B4, j)$ where j is a multiple of 1 (and in the iteration range) are in the preserved set for $B4$.

We now define *synchronization cycles* and state the relevant related results.

Definition 5 *A sequence of blocks in the SCG* $\{b_{j_0}, b_{j_1}, ...b_{j_{n-1}}\}$ *constitutes a* **Synchronization Cycle** *if there exist* $\{\vec{\delta_{j_0}}, \vec{\delta_{j_1}}, ...\vec{\delta_{j_{n-1}}}\}$ *such that* $(j_i, \vec{\delta_{j_i}}) \in Preserved(j_{(i+1)\, modulo\, n})$ *and there is an edge in the SCG from* b_{j_i} *to* $b_{j_{(i+1)\, modulo\, n}}$.

A cycle in the *SCG* is a synchronization cycle, if for every edge in the cycle, the source block is in the preserved set of the sink block, with some distance vector. If b_j is member of a synchronization cycle, then $(b_j, \vec{\delta})$ belongs to the preserved set of b_j, where $\vec{\delta}$ is the sum of distance vector terms around the cycle. Converse of this statement can be argued in an analogous manner. This yields the following theorem:

Theorem 6 *A block b_m in the SCG is a member of some synchronization cycle if and only if there exists a $\vec{\delta}$ such that $(m, \vec{\delta}) \in Preserved(m)$.*

The above theorem provides a method of identifying synchronization cycles during the computation of preserved sets. We introduce a representation for a class of distance vectors to have the ability to capture and represent the effect of such synchronization cycles.

$$\vec{\delta^*} = \{n\vec{\delta}\text{: } n \text{ is a positive integer}\}$$

$$(i, \vec{\delta^*}) = \{(i, \vec{\delta})\text{: } \vec{\delta} \in \vec{\delta^*}\}$$

Synchronization distances can be repeatedly added over a synchronization cycle. This is stated in the following theorem:

Theorem 7 *For any block b_i in the SCG $(i, \vec{\delta^*}) \subseteq Preserved(i)$ if $(i, \vec{\delta}) \in Preserved(i)$.*

The data flow framework described earlier in this section can be modified to detect synchronization cycles in a straightforward manner using Theorem 6 and add all entries to SCPreserved sets that are a consequence of synchronization cycles compactly using Theorem 7. The *union* and Addedge functions have to understand the meaning of the entries corresponding to synchronization cycles and know how to combine them.

6 An Algebraic Formulation

In a large class of programs, the problem of verifying whether a dependence is preserved can be transformed to the problem of determining whether there exists a non-negative integral solution to a system of linear equations. The transformation procedure takes nearly linear time and the method can be used to get precise (up to symbolic execution) results. Although the general problem of solving such a system of equations is known to be NP-hard, the actual systems generated are expected to be small and fairly easy to solve.

We illustrate this method here, which is faster than the method discussed earlier. For this method to be applicable to a program, there can be only one `POST` instruction for each event variable, and conditional control flow must be internal to the blocks in the *SCG*. However, multiple `WAIT` instructions on the same event variable are permitted and parallel case and loop style control flow may be present in the *SCG*.

These restrictions allow us to compute exact preserved dependence information, unlike the approximate solution technique presented in the last section.

If there is a control flow edge from b_p to b_q with distance $\vec{s}$, we can infer that b_p, $\vec{s}$ distance away *must* finish execution before b_q with a given iteration vector is executed, assuming the instances of b_p and b_q in question are executed. The same result holds if there is a synchronization edge from b_p to b_q, and this is the only synchronization edge incident on b_q (which must be the case if there is only one `POST` per event variable). In the absence of control flow in the *SCG*, this translates to the fact that if there is a path from b_p to b_q of total distance $\vec{d}$ in the *SCG*, then b_p, distance $\vec{d}$ iterations away must be executed before b_q with a given iteration vector is executed, if both are executed. Since the *SCG* is assumed to be the only information available about the program, we have the following theorem:

Theorem 8 *A dependence from a block b_p to block b_q with distance $\vec{\delta}$ can be proved to be preserved if and only if there exists a path from b_p to b_q of distance $\vec{\delta}$ in the synchronized control flow graph of the program.*

A technique developed by Tarjan [Tar81] can be used to find a canonical regular expression representing all paths between any two nodes in a graph in nearly linear time. Finding whether a particular value of distance corresponds to a member of the regular expression can be transformed to finding whether a non-negative solution exists to any of a set of simultaneous linear equation systems. We shall illustrate this with an example.

Figure 6 shows the synchronized control flow graph of a code section inside a loop nest. The set of paths from b_p to b_q in the Figure 6 is represented by the regular expression:

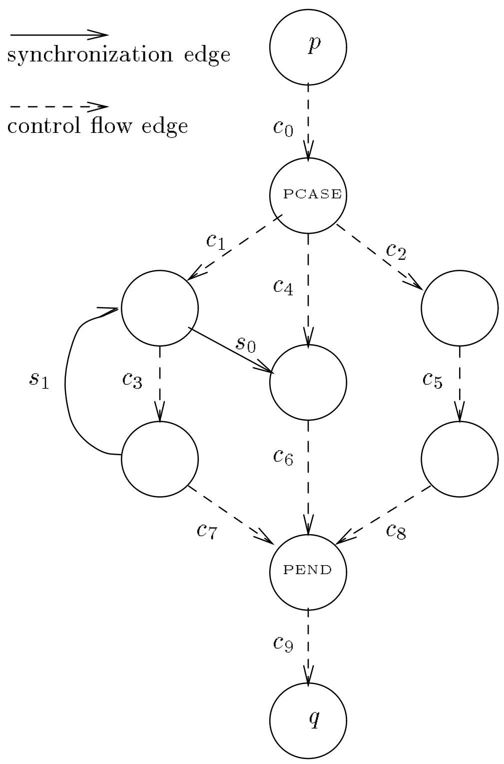

Figure 6: Set of paths from node p to node q in the *SCG* are represented by: $c_0c_1c_3(s_1c_3)^*c_7c_9 \cup c_0c_1(c_3s_1)^*s_0c_6c_9 \cup c_0c_4c_6c_9 \cup c_0c_2c_5c_8c_9$

$$p \rightarrow q = c_0c_1c_3(s_1c_3)^*c_7c_9 \cup c_0c_1(c_3s_1)^*s_0c_6c_9 \cup c_0c_4c_6c_9 \cup c_0c_2c_5c_8c_9$$

Since the control flow edges do not carry any distance, an equivalent regular expression for the purpose of analyzing distances is:

$$p \rightarrow q = s_1^* \cup s_1^* s_0$$

Let the edges s_1 and s_2 have synchronization distance vectors $\vec{S_1}$ and $\vec{S_2}$ associated with them. Let $\vec{\delta}$ be the distance of a dependence from b_p to b_q. By theorem 8, this dependence is preserved if and only if there is distance $\vec{\delta}$ path from b_p to b_q. Thus, this dependence is preserved if and only if there exist non-negative integral solutions to either of the following two systems of linear equations:

$$I_1\vec{S_1} = \vec{\delta} \quad (5)$$

$$I_2\vec{S_1} + \vec{S_2} = \vec{\delta} \quad (6)$$

Here I_1, I_2 are integer variables.

Solving a system of equations in non-negative integers is known to be NP-hard. However, this formulation would normally lead to a small system of equations. Moreover, the coefficients of individual terms relate to components of distance vectors which are normally small integers. Thus any practical strategy for solving such systems would yield good results.

7 The `CLEAR` statement

Up to this point, we have assumed the absence of the `CLEAR` statement, which has the effect of resetting the value of an event variable to *clear*. In this section we discuss how the `CLEAR` statements affect program synchronization behavior and its relevance to our analysis.

The `CLEAR` statement is commonly used as a way to reuse an event variable for synchronization. If `CLEAR` statements are ignored for identifying potential data races, all true data races will still be recognised. However, some of the potential data races can be proved to be non-existent by analyzing the `CLEAR` statements. Thus, analysis of `CLEAR` statements is a critical optimization for our methods to work effectively.

In our synchronized control flow graph, we have an edge from every `POST` statement on an event variable to every `WAIT` statement on the same variable, implying that any of the `POST` statements can set the value of the event variable to *posted* and enable execution past the `WAIT` statement. However, if we can prove that a `CLEAR` statement must execute before a `WAIT` statement and after a `POST` statement (if both are executed), that particular `POST` statement cannot be responsible for triggering the execution at that `WAIT` statement, and hence the synchronization edge between this `POST-WAIT` pair can be removed. This makes the analysis using the *SCG* more accurate[§].

Let `ev` be a scalar event variable. If there is at least one control flow path from a `POST(ev)` statement to a `WAIT(ev)` statement, and every control flow path from the `POST(ev)` to the `WAIT(ev)` statement has a `CLEAR(ev)` statement in it, the synchronization edge between the `POST(ev)` and the `WAIT(ev)` can be removed. This is illustrated in Figure 7.

While the above is a sufficient condition to enable removing an edge from the *SCG*, it is not necessary since it relies only on control flow information. Synchronization mechanisms themselves can provide additional information which can help remove synchronization edges in the *SCG*. We are investigating ways in which handling of `CLEAR` statements can be integrated with the rest of the analysis.

[§]Inside loops, analysis of `CLEAR` statements can often help attach more precise synchronization distance vectors, which is in effect analogous to reducing the number of `POSTS` that can trigger a `WAIT`

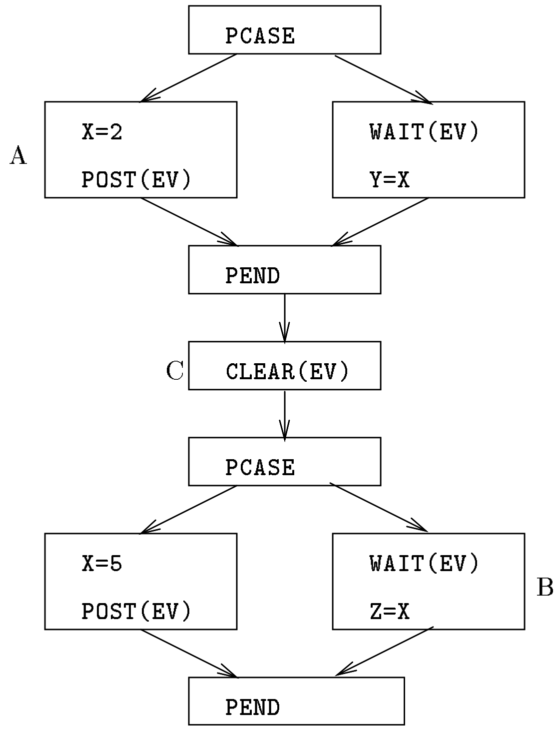

Figure 7: A program using a `CLEAR` statement. The synchronization edge from block A to block B can be eliminated due to the `CLEAR` statement in block C.

8 Conclusions

Although controlled non-determinism can be an important feature of an efficient parallel program, inadvertent non-determinism can be a source of hard to find errors in parallel programs. The problem addressed in this paper is to statically verify which data dependences in a program are potential sources of non-determinism, in the presence of event variable synchronization. Our experience with parallel programming tools suggests that this is an important problem. We have implemented a test for *preserved* dependences based on some of the ideas contained in this paper in ParaScope, a parallel programming environment being developed at Rice. Our preliminary experience suggests that the automatic assistance in judging the effect of event variable synchronization is useful even in fairly simple applications.

We have restricted our attention to event synchronization, but our methods can be directly applied to other synchronization mechanisms that restrict the possible orders of execution between certain statement pairs. These include ordered critical sections, ADA style rendezvous and some types of semaphores. In all these, the *SCG* can be constructed from the program text and rest of the analysis is identical. Our methods are not useful for synchronization mechanisms that enforce some kind of mutual exclusion, but no statement execution order. These include locks and critical sections.

In the paper we assume that a sequential execution of a program is defined (program is serializable) and we test for *correctness*, that is whether the parallel version of the program is going to compute the same results as the sequential execution. While this was considered useful in reasoning about parallel programs and meaningful for many parallel applications, it is as easy to test for non-determinism using the methods developed.

We also assumed that distance vectors for all dependences are integer vectors and computable by data dependence analysis. The methods can actually be applied in the presence of partial distance information in many cases. However, the precision of the methods will be limited by the knowledge about distance vectors.

The principal strength of our approach is that it can be applied to parallel programs with task and loop style parallelism. Nested loops and dependences across loop nests can be analyzed in a natural way.

References

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AM85] W. F. Applebe and C. E. McDowell. Anomaly reporting — a tool for debugging and developing numerical algorithms. In *First International Conference on Supercomputers*, Florida, December 1985.

[AP87] Todd Allen and David Padua. Debugging FORTRAN on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, September 1987.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BBC+88] V. Balasundaram, D. Bäumgartner, D. Callahan, K. Kennedy, and J. Subhlok. PTOOL: A system for static analysis of parallel programs. Rice COMP TR88-71, Dept. of Computer Science, Rice University, June 1988.

[BK89] V. Balasundaram and K. Kennedy. Compile time detection of race conditions in a parallel program. In *Proceedings of the International Conference on Supercomputing*, Crete, Greece, 1989.

[CK87] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, 1987. Available as Rice University, Department of Computer Science Technical Report TR87-56, July 1987.

[CS88] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison,WA, May 1988. Available as Rice University, Dept. of Computer Science Technical Report TR88-70.

[EP88] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison,WA, May 1988.

[FJS85] P. O. Frederickson, R. E. Jones, and B. T. Smith. Synchronization and control of parallel algorithms. *Parallel Computing*, (2):255–264, 1985.

[Hen87] L. Henderson. The usefulness of dependecy-analysis tools in parallel programming: Experiences using PTOOL. Technical Report Preprint LA-UR-87-3135, Los Alamos National Laboratory, September 1987.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eigth ACM Symposium on the Principles of Programming Languages*, pages 207–218, Williamsburgh, VA, January 1981.

[Kuc78] D. Kuck. *The Structures of Computers and Computation.* John Wiley and Sons, New York, 1978.

[MC88] B. P. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, pages 135–144, Atlanta,GA, June 1988.

[Par88] The Parallel Computing Forum. *PCF Fortran: Language Definition*, 1 edition, August 1988.

[Sch89] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation*, pages 285–297, Portland, OR, June 1989.

[Tar81] R. Tarjan. Fast algorithms for solving path problems. *JACM*, 28(3):594–614, July 1981.

[Tay83] R. N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Wol82] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.