Event Synchronization Analysis for Debugging Parallel Programs *

Perry A. Emrath Sanjoy Ghosh
David A. Padua
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
305 Talbot Laboratory
104 South Wright Street
Urbana, Illinois 61801

Abstract

One of the major difficulties of explicit parallel programming for a shared memory machine model is detecting the potential for nondeterminacy and identifying its causes. There will often be shared variables in a parallel program, and the tasks comprising the program may need to be synchronized when accessing these variables.

This paper discusses this problem and presents a method for automatically detecting non-determinacy in parallel programs that utilize event style synchronization instructions, using the Post, Wait, and Clear primitives. With event style synchronization, especially when there are many references to the same event, the difficulty lies in computing the execution order that is guaranteed given the synchronization instructions and the sequential components of the program. The main result in this paper is an algorithm that computes such an execution order and yields a Task Graph upon which a nondeterminacy detection algorithm can be applied.

We have focused on events because they are a frequently used synchronization mechanism in parallel versions of Fortran, including Cray [Cray87], IBM [IBM88], Cedar [GPHL88], and PCF Fortran [PCF88].

[**Keywords**: Debugging, Nondeterminacy, Parallel Programming, Parallel Processing, Race Detection, Software Tools, Synchronization]

*This work was supported in part by the **National Science Foundation** under Grant No. US NSF MIP84-10110, the **US Department of Energy** under Grant No. US DOE DE FG02-85ER25001, and a donation from the **IBM Corporation**, and Concurrent Computer Corporation.

1 Introduction

As the potential of parallel processing is being realized through the development of academic and commercial parallel processing systems, it is also becoming apparent that writing and debugging parallel programs is considerably harder than writing and debugging sequential programs. The principal reason for this is that, unlike sequential programs where ordering between accesses to the same memory location are guaranteed by the sequential semantics, in parallel programs this ordering can sometimes only be guaranteed by synchronization instructions. Incorrect orderings, for example, due to incorrect synchronization, typically result in incorrect output of the program.

A *determinate* program is one which will always produce the same output on every run[1]. Programs executing sequentially are inherently determinate. The tasks[2] comprising a nondeterminate parallel program may execute and access shared memory in different orders on different runs. A *race condition* is the source of nondeterminacy and occurs when there are no synchronization operations that guarantee ordering among two or more references to a shared memory location, at least one of which is a write. For example, consider the following program.

Program 1

```
a = 5
cobegin
   a = 1
```

[1] By multiple runs of a program we mean multiple executions where both the code and the input are unchanged.

[2] By a *task* we mean a single thread of execution.

```
//
   b = a - 3
   if (b>0) then c = 5
   else c = 1
coend
print *,b,c
```

The output of this program could either be b = 2, c = 5; or b = -2, c = 1 because of the race between the references to a in a = 1 and b = a - 3.

This paper is a report on a continuing research effort at the University of Illinois [AlPa87], [EmPa88], the goal of which is to develop a debugging tool for use on programs originally written as parallel programs[3]. The tool is intended to detect race conditions through a combination of static (compile-time) and trace (run-time) analysis. Communication between asynchronous tasks takes place through a shared address space and by the use of explicit synchronization instructions. The current implementation is designed for use with Cedar Fortran [GPHL88], a Fortran dialect that runs on the Cedar multiprocessor [KDLS86] under development at the University of Illinois. However, the approach described is applicable to languages other than Cedar Fortran.

2 Race Detection

We describe our approach to the detection of races using a combination of static and trace analysis. In many cases it is possible to detect races by static analysis alone, which resembles analysis for data dependences [Bane88]. Statements that execute in parallel and access the same location are identified. If there is no guaranteed ordering between two or more of these instances, of which at least one is a write, then a race has been detected [EmPa88].

In cases similar to that of Program 1 above, complete compile-time detection of nondeterminacy is easy. In more complicated cases involving loops and arrays, subscript analysis [Bane88] might be necessary.

Analogously to situations where dependence analysis fails to give a definite dependence relation between two statement instances[4], static analysis for race detection may fail to give a definite answer. For example, consider the following program:

[3] We distinguish this situation from that of parallel programs derived by a program restructurer. A restructurer should always produce determinate programs if the source program is sequential.

[4] A statement that occurs within a loop may be executed several times. Such an execution is known as a *statement instance*.

Program 2

```
      read *,K
      doall i = 1,N
S:       A(K(i)) = ...
      end doall
```

In this case, the existence of a race between different instances of S depends on the values of K, which are not known until run-time. In this case, a *potential race* is assumed between instances of S. A similar situation arises when the subscripts involved are nonlinear, since there are no efficient algorithms for precise dependence analysis of nonlinear subscripts.

One way to overcome the above shortcomings of static analysis is to use trace analysis. Trace analysis works by first obtaining an memory access trace of the program for a set of input values. It then analyzes this trace output for races using algorithms described below.

Therefore, our approach to detecting races in a parallel program has two stages. In the first stage we perform static analysis to determine as many races as we can. In addition, we may also find potential races that are impossible to resolve without trace analysis. At this stage the user may be satisfied with the imprecise information available. Otherwise, the trace analyzer may be used to detect races for specific input values. Trace analysis is slow and is valid only for the particular data input used.

Thus our work concerns the area of program debugging since the errors detected are only those that arise for the input set being used to test the program.

3 Task Graphs

In this paper we concentrate on trace analysis. For this purpose, a graph is constructed from the significant events that are recorded in a log file by executing an instrumented version of the program being debugged. The arcs in the graph specify guaranteed run-time ordering between these events. The recorded events include shared memory references, task initiations and terminations, task waits, and the execution of synchronization instructions. The information contained in the log file includes the operation performed, task number, program line number, value of the variable, address, etc.

Cedar Fortran has several ways to achieve parallel execution. One way is to use the `ctskstart()` and `ctskwait()` instructions [GPHL88]. The instruction `tskid = ctskstart(subr)` spawns a new task to execute subroutine `subr` and returns an integer

to identify it in `tskid`. `Call ctskwait(tskid)` suspends the calling task until the task whose identifier is `tskid` completes execution.

Parallel execution can also be achieved by the `cdoall` and `cdoacross` loops, which may execute iterations of the loop body in parallel. However, for the trace log, each iteration behaves exactly like a task created by an implicit `ctskstart()` at the beginning of the `cdoall`. Therefore, in this report we do not distinguish between tasks started by a `ctskstart()` and an iteration of a `cdoall` or `cdoacross`.

In addition, synchronization instructions can be used to order points in asynchronous tasks. Hence, we identify four types of ordering in the execution of a parallel program. The first three types of ordering are collectively called *task ordering.*

- Task Sequential Ordering. This ordering is due to the sequential execution of instructions belonging to the same task.
- Task Start Ordering. This ordering is due to the fact that every instruction in a started task follows the `ctskstart()` instruction that created it.
- Task Wait Ordering. This ordering is due to the fact that a `ctskwait()` instruction follows every instruction in the waited for task.
- Synchronization Ordering. This ordering is due to synchronization instructions placed in the program.

A TASK GRAPH is a graph with each node representing a trace item. Corresponding to the four categories of ordering there are four categories of directed arcs in the TASK GRAPH:

- Machine Arc[MiPa87]. This is an arc from a trace item to the immediately following trace item in the same task.
- Task Start Arc. This is an arc from a Task Start node to the first node of the started task.
- Task Wait Arc. This is an arc from the last node of a task being waited upon to the Task Wait node.
- Synchronization Arc. This is an arc between nodes that are ordered by synchronization instructions.

Machine arcs, Task Start arcs, and Task Wait arcs, which are collectively known as *task arcs,* are very easily generated from the trace log. Generating synchronization arcs can be noticeably more difficult depending on the type of synchronization. Sections 4, 5, and 6 deal with this problem for event (i.e., post and wait) synchronization.

Our approach to detecting races in the trace consists of two steps. The first step is to determine the timing relationships among trace items as accurately as possible. This is done by constructing a TASK GRAPH with the trace items as nodes and a path from each node to every node that is guaranteed to follow it in time. We start by inserting the task arcs first and then using an iterative algorithm to add synchronization arcs. The second step is to detect races within this graph. This step is conceptually straightforward since any pair of references to the same shared memory location (not both reads), such that there is no path from one to the other, constitutes a race.

The number of races detected in a program could be very large and easily overwhelm the user, therefore they should be reported in a user-friendly manner. For instance, a `cdoall` with a loop count of 1000 might generate 999 races between successive iterations of the loop. Instead of deluging the user with that many messages, the races could be "folded" into one message and the race between adjacent iterations of the `cdoall` presented in a compact fashion.

A race may cause other races to either manifest themselves or not, in a random manner. [AlPa87] defines the "hides" relationship between races. Two races are related by the "hides" relation if the outcome of the first can affect the occurrence of the second. For example, consider the following program:

Program 3

```
i = 10
cobegin
   a = 5
//
   a = 10
coend
if (a .eq.  10) then
   cobegin
      i = 0
   //
      i = 1
   coend
end if
```

In this program the race on a is related to the race on i by the "hides" relation. Whether the race on i occurs depends on the outcome of the race on a. At least one race from a set of races related by the

"hides" relation will always show up in a trace. In the previous example, even if the race on i is hidden by the race on a, the race on a will always show up. Thus, iterative use of trace analysis may be required to remove all races.

As an illustration of the above ideas we use Program 4 below. The parallel construct in this program is the cdoacross loop in which the iterations are executed in parallel, and cascade synchronization, from lower to higher iterations, is allowed. This synchronization may be achieved by means of the advance() and await() [Alli85] instructions. Advance(x) waits until the value of x is equal to the current iteration count and then increments x by 1. Await(x,y) waits until the value of x plus the value of y is greater than or equal to the current iteration count. The value y therefore represents the offset back to the iteration upon which the current iteration should wait.

Program 4

```
cdoacross i = 1,4
   a(i+1) = b(i) + 1.0
   call advance(x)
   c(i) = a(i) + 2.0
end cdoacross
```

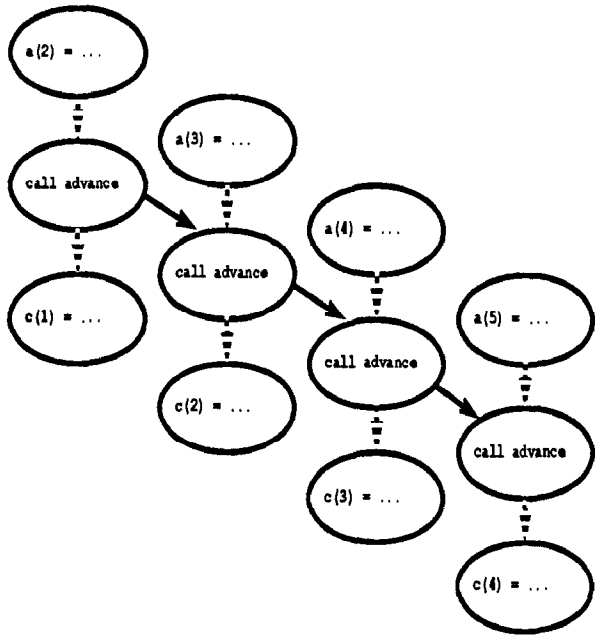


Fig. 1. Example of TASK GRAPH

Fig. 1 shows the TASK GRAPH generated from it. In the TASK GRAPHs presented in this paper, machine arcs, task start arcs, and task wait arcs are all represented by dashed arrows. Synchronization arcs are represented by solid arrows. The synchronization arcs between iterations are inserted because the advance() instruction has an implicit wait until the previous iteration has executed the same instruction. This forces the third instruction in iterations 2, 3, and 4 to wait until the second instruction in iterations 1, 2, and 3, respectively, have been executed. To check that this program has no races, it is enough to observe that for any pair of references to the same memory location, such as a(3), there is a path from the write of a(3) to the read of the same variable. This means that the read always follows the write, and therefore there is no race.

Before continuing with the following sections, some definitions will be needed. Given a directed acyclic graph $G = (V,E)$ consisting of a set of nodes V and a set of arcs E, we say that a node a is a *common ancestor* of N (a subset of V) if there is a path from a to each and every node in N. We say that a node c is a *closest common ancestor* of N if there is no path from c to any other common ancestor of N. Note that N may have more than one closest common ancestor. A *root node* of G is any node that has no ancestors.

4 Identifying Synchronization

There are many synchronization mechanisms, for example, *semaphores*, *locks* (to implement mutual exclusion), and *events*. We have already described one of them, the advance() and await() instructions. Events are considerably more difficult to analyze than advance() and await(), as we shall describe below. The remainder of this paper deals with the problem of generating the synchronization arcs corresponding to event synchronization.

The principal synchronization mechanism we consider in this paper is the event. An event has two states: *posted* and *cleared*. A posted event usually signifies that some action has been performed, whereas a cleared event signifies the opposite. Tasks can perform the following operations on an event:

- POST. This sets the state of the event to posted. It has no effect if the event is already posted.
- WAIT. This suspends the calling task until the state of the event changes to posted. If the event is already posted the task proceeds without pause.
- CLEAR. This resets the state of the event to cleared.

The complete TASK GRAPH for a program utilizing event type synchronization consists of machine arcs, task start arcs, task wait arcs, and synchronization arcs. The machine arcs, task start arcs, and task wait

arcs are easy to insert from the program trace. The difficult job is to identify the synchronization arcs. Specifically, the problem is to compute for each event Wait w, the set of nodes in the graph that always precede w due to synchronizations.

Each Wait w has to be preceded by at least one Post on the same event and that event must not be cleared before w is executed. So, Posts that either always follow w, or are always cleared before w is executed, cannot trigger w[5]. Any other Posts on the same event may trigger w. If there is more than one such Post, there is no way to decide which of them actually triggered (or might trigger) w. However, a node that precedes *all* of those Posts necessarily precedes w.

Every common ancestor of this set of Posts satisfies this requirement, and therefore precedes w. Instead of the complete set of common ancestors, what is desired is the set of closest common ancestors. This, as we said in Section 2, is the set of common ancestors such that any other common ancestor precedes at least one of them. Once the set of closest common ancestors has been found, synchronization arcs can be added to the TASK GRAPH from each closest common ancestor to w. This justifies the following set of rules:

- A Wait on event E can be triggered by any Post on the same event E, if that Post does not follow the Wait and is not cleared before the Wait.

- Synchronization arcs are added from the closest common ancestors of the Posts that might trigger the Wait to the Wait.

If a Post on E is Cleared before the Wait, then it follows that the Post must precede the Wait, but it is still true that this Post cannot trigger the Wait, so some other Post must trigger it.

As an example of the application of these rules consider the TASK GRAPH in Fig. 2. Representations of TASK GRAPHs in this paper only show nodes that represent operations on events, task starts, or task waits. Every node in the TASK GRAPH is assigned a unique number to identify it. Numbers are assigned so that for a pair of nodes that are task ordered, the node that executes later has a higher number. Each node shows the number, the operation, and the event name if it is an event operation.

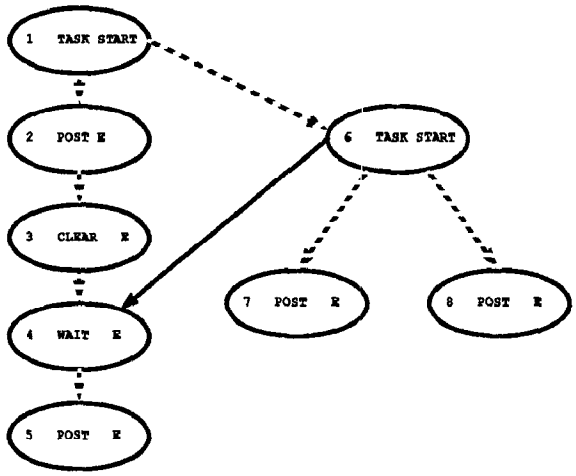


Fig. 2. Example of the synchronization rules

In this graph it is clear that the Post in node 2 cannot trigger the Wait in node 4 because it is cleared by the Clear in node 3 before the Wait is executed. Similarly the Post in node 5 cannot trigger the Wait because it is executed after the Wait is triggered. But the Posts in nodes 7 and 8 have nothing to prevent them from triggering the Wait. There is no way to decide which of the two Posts actually triggers the Wait. Therefore the only certainty is that the Wait proceeds after the closest common ancestor of the two Posts, which is the Task Start in node 6. Therefore a synchronization arc is present from node 6 to the Wait in node 4.

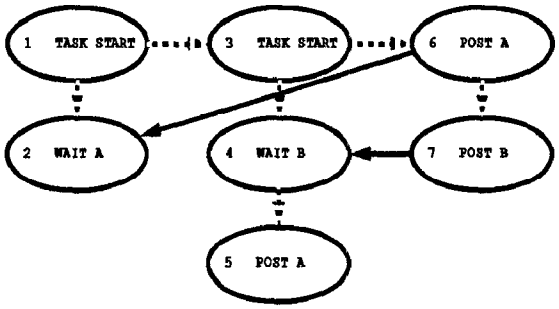


Fig. 3. Example of TASK GRAPH

The presence of more than one event makes iteration essential because an event may influence the orderings between Posts and Waits on other events. For example consider the TASK GRAPH in Fig. 3. If the Wait on event A in node 2 is examined first, both the Posts in nodes 5 and 6 seem capable of trigger-

[5] A Wait w is *triggered* when some Post on the same event is executed, thus allowing the task that executed w to proceed.

ing it. However, when the Wait on event B in node 4 is examined it is clear that a synchronization arc is needed from node 7 to node 4. It becomes clear that the Post in node 5 follows the Post in node 6. Therefore, we know that the Wait in node 2 always follows the Post in node 6. That is the reason for the presence of a synchronization arc from node 6 to node 2.

5 Algorithm for Adding Synchronization Arcs

We begin with an approximation to the TASK GRAPH constructed from the program trace output. It includes all arcs except synchronization arcs. Procedure SYNC, shown below, then iteratively adds synchronization arcs to the TASK GRAPH. As long as it finds new arcs to add, it checks every Wait w. For each Wait w, it performs a sequence of three steps.

(1) It computes the set of Posts that might trigger w. This is the set of all Posts on the same event minus those Posts that either follow w or are cleared before w is executed. This set might contract from iteration to iteration due to new synchronization arcs.

(2) It then computes the set of closest common ancestors for this set of Posts.

(3) Finally, an arc is added from every member of this set of closest common ancestors to w.

Procedure SYNC uses two predicates, isCleared and Follows, as well as two procedures, AddArc and CloseCommAnc. IsCleared(p,w) is a predicate that is true only if there is a Clear on event E, on a path from node p to node w, where node p is a Post on E and w is a Wait on E. Follows(m,n) is a predicate that is true if there is a path from node m to node n in the current version of the TASK GRAPH. AddArc(N,n) is a procedure that inserts arcs from every member of the set of nodes N to node n. If an arc does not already exist, it adds one and also sets the flag *change* to true. CloseCommAnc(N) is a procedure that computes the closest common ancestors of the set of nodes N, for the current approximation to the TASK GRAPH.

```
procedure SYNC
while change do
   change ← FALSE
   for each Wait w do
      TRIG ← {}
      for each Post p on same event as w do
         if (¬ isCleared(p,w) ∧ ¬ Follows(w,p))
            TRIG ← TRIG ∪ {p}
      AddArc(CloseCommAnc(TRIG),w)
```

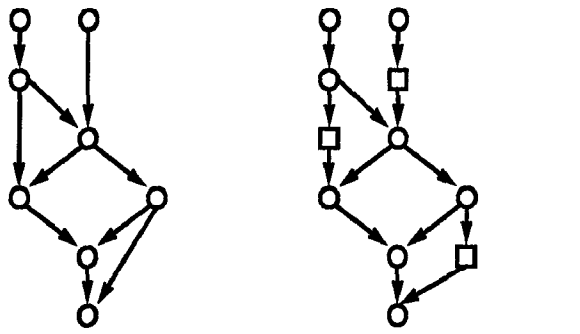
Termination of procedure SYNC is guaranteed by the fact that the iteration of the **while** loop stops as soon as it does not find any new arcs to add. At no point does it delete an arc, thus precluding the possibility of oscillations. Since the number of nodes in the TASK GRAPH is finite, the maximum possible number of arcs is also finite.

CloseCommAnc is an algorithm that computes the closest common ancestors of a subset N of nodes in a directed acyclic graph (DAG). For simplicity of explanation, we assume that the input to the algorithm is a *normalized* DAG. We say that a DAG, $G = (V,E)$, is *normalized* if $<m,n> \in E$ implies that level(n) - level(m) = 1. The level of a node n in G, denoted level(n), is defined as

$$\text{level}(n) = 0, \quad \text{if } n \text{ is a root node}$$
$$\text{level}(n) = \max_{<m,n>\in E} (\text{level}(m)) + 1, \quad \text{otherwise}$$

where $<m,n>$ is an arc from node m to n in G.

Any DAG can be transformed into normalized form by adding dummy nodes. If there are arcs of the form $<m,n> \in E$ with level(n) > level(m) + 1, we could replace these arcs with paths containing level(n) - level(m) - 1 nodes. An example of normalization is shown in Fig 4.



Unnormalized DAG

Normalized DAG.
(The □ s are the dummy nodes)

Fig. 4. Example of DAG Normalization

The algorithm to compute the closest common ancestors is given below. It uses the function Parent which operates on a set of nodes S, and returns the set

of nodes that immediately precede any of the nodes in S. More formally, Parent(S) = { m | <m,n> $\in E$, for some $n \in S$}. Parent^k(S), when $k > 0$, denotes the result of k applications of Parent, and Parent^0(S) = S.

The algorithm starts by computing k, the lowest level of the nodes in the argument set N (i.e., closest to the root). Then the algorithm (in the first **for** loop) computes the ancestors at level k for each node in N. These sets are stored in the array *Ancestors*[]. If for some $n \in N$, level(n) = k, then *Ancestors*[n] = {n}, the set containing n itself.

The algorithm proceeds iteratively, on each iteration processing the sets of ancestors of the nodes in N at a given level. Thus, the first iteration processes the ancestors at level k, the second iteration, the ancestors at level k-1, and so on.

In general, on the iteration processing the ancestors at level l ($l \leq k$) the algorithm first computes *New_Closest_CA*, the intersection of the sets of ancestors of the nodes in N at level l. Each member of the intersection is a closest common ancestor. The intersection is accumulated in the variable *Closest_CA* that will eventually contain all closest common ancestors.

The Parents of the members of the intersection are accumulated into the variable *Other_CA*, so that at the end of the iteration processing level l, *Other_CA* contains the set of nodes at level $l - 1$ that are common ancestors of N, but not closest common ancestors. Finally, preparation is made for the next iteration by computing the *Ancestors* of each original node at the next lower level, but without including the ancestors of those closest common ancestors already in variable *Closest_CA*. At the top of each iteration, all the members of *Ancestors*[] and *Other_CA* are at the same level in the graph.

```
procedure CloseCommAnc (N)
k ← min (level(n))
    n∈N
for each n in N
    Ancestors[n] ← Parent^(level(n)−k)({n})
Closest_CA ← {}
Other_CA ← {}
while (all Ancestors[n] ≠ {})
    New_Closest_CA ← ⋂ Ancestors[n]
                     n∈N
    Closest_CA ← Closest_CA ∪ New_Closest_CA
    Other_CA ← Parent(Other_CA) ∪
        Parent(New_Closest_CA)
    for each n in N do
        Ancestors[n] ← Parent(Ancestors[n]) -
            Other_CA
return(Closest_CA)
```

6 Example of TASK GRAPH generation

As an example of the above approach, consider the graph in Fig. 5. The goal is to compute for each Wait, the nodes in the program that necessarily precede it.

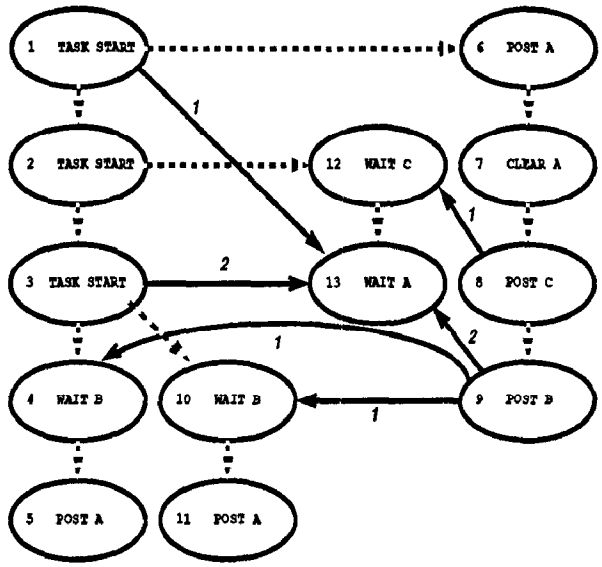


Fig. 5. Illustration of the SYNC algorithm

To add synchronization arcs, the procedure SYNC is applied to the graph. On each iteration of the **while** loop in SYNC, we assume that the Waits are considered in ascending order of event name. Furthermore, for Waits on the same event, a Wait w that always executes before another Wait y is always considered before y. Therefore, SYNC applied to the given TASK GRAPH operates as follows. (Each synchronization arc is labelled with the iteration number that added it.)

The procedure first looks at node 13. The Posts on event A are nodes 6, 5, and 11. The only common ancestor of these Posts is the Task Start in node 1 and so an arc is added from node 1 to node 13. It looks at the Wait in node 4 next. The only Post on event B is node 9. Therefore, an arc is added from node 9 to node 4. For the same reason, it adds an arc from node 9 to node 10. Finally, it adds an arc from node 8 to node 12. The first iteration of the **while** loop is over. Since new arcs have been added another iteration of the **while** loop is performed.

On the second iteration, when node 13 is examined it finds that the Post in node 6 has a path to node 13, via nodes 7, 8, and 12. But this path contains a Clear on event A in node 7. Therefore, node 6 cannot trigger node 13. So the set of nodes that might

trigger node 13 contains only nodes 5 and 11. To find the closest common ancestors of this set the graph is traversed backwards, finding two closest common ancestors. One is the Task Start in node 3, and the other is the Post in node 9. Arcs are added from nodes 3 and 9 to node 13. This iteration then completes without any further changes to the graph. Another iteration is performed because new arcs were added. On the third iteration no new arcs are added, so SYNC terminates.

Now we have a complete TASK GRAPH and can apply the race detection algorithm to locate any races that exist. This involves examining the TASK GRAPH, looking for references to the same variable (not all reads) that are not connected by some path. With no path, the execution order of the two references is not guaranteed. Any races that are found are reported to the user.

7 Related Work

Other groups have also studied the problem discussed in this paper. In [CaSu88] a data flow formulation is presented for race detection by static analysis. This formulation does not deal with loops. The principal synchronization primitive considered is the event, which is one of the two primitives we handle. However, they do not handle the Clear operation.

In [MiCh88] a system is described that aids the user in debugging parallel programs. While the paper discusses synchronization edges and their use in detecting races, it does not describe any procedure to compute these edges. We feel that this is the main contribution of our paper.

In [Scho89] a procedure is outlined for detecting races on the fly. This procedure only handles pairwise synchronization operations.

8 Conclusion

In this paper we have described a system that automatically detects races in a parallel program. The Cedar Fortran compiler is used to insert instructions that write trace information to a file. This processed program is then compiled and executed, producing a dynamic execution trace of the program. The trace output has all the information needed to build a TASK GRAPH. The trace also logs points of event style synchronization. We have demonstrated an algorithm that can identify which nodes in the TASK GRAPH are synchronized by event operations.

An advantage of our approach is that it does not place any restriction on the control structure of the programs. Iterative control structures, such as loops and subroutines, that are difficult to analyze statically, can easily be traced. Analysis of dynamic traces was used because it eliminates uncertainties about control flow that are present when static analysis is used exclusively. Also, actual scheduling of the program (i.e., the parallel tasks) does not affect the results of the analysis. In fact, the tasks can be run sequentially on a uniprocessor to generate the trace, and races will still be correctly detected.

There are several avenues for future work. We are studying ways to improve the efficiency of the algorithm. Several optimizations are possible that will help speed up the algorithm. This will become more important as we tackle larger parallel programs. The implementation presently is restricted to synchronizations using events. We believe the algorithm is extendable to several other commonly used synchronization mechanisms such as binary and counting semaphores, lock-unlock, the Cedar synchronization primitives, etc.

Much of this work is currently in progress. The tracer, the SYNC procedure, and race detector have been implemented, and some experiments have been performed with it to demonstrate correct operation and the practicality and usefulness of the approach.

References

[Alli85] *FX/Series Architecture Manual Concurrency Supplement*, Alliant Computer Systems Corp., April 1985.

[AlPa87] Todd R. Allen, and David A. Padua. *Debugging Fortran on a Shared Memory Machine*, Proceedings of the 1987 International Conference on Parallel Processing, pp. 721-727, Aug. 1987.

[Bane88] Utpal Banerjee. *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[CaSu88] David Callahan, and Jaspal Subhlok. *Static Analysis of Low-level Synchronization*, Proceedings of the Workshop on Parallel and Distributed Debugging, pp. 100-111, May 1988.

[Cray87] *Cray X-MP Multitasking Programmer's Reference Manual*, Cray Research, Inc., 1987.

[EmPa88] Perry A. Emrath, and David A. Padua. *Automatic Detection of Nondeterminacy*

in Parallel Programs, Proceedings, Workshop on Parallel and Distributed Debugging, pp. 89-99, May 1988.

[GPHL88] M. D. Guzzi, D. A. Padua, J. P. Hoeflinger, and D. H. Lawrie. *Cedar Fortran and Other Vector and Parallel Fortran Dialects*, Proceedings SUPERCOMPUTING '88, pp. 114-121, Nov. 1988.

[IBM88] *IBM Parallel FORTRAN Language and Library Reference*, IBM Corp., March 1988.

[KDLS86] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh. *Parallel supercomputing today and the Cedar approach*, Science, vol. 231, pp. 967-974, Feb. 1986.

[MiCh88] Barton P. Miller, and Jong-Deok Choi. *A Mechanism for Efficient Debugging of Parallel Programs*, Proceedings of the Workshop on Parallel and Distributed Debugging, pp. 141-150, May 1988.

[MiPa87] S. P. Midkiff, and D. A. Padua. *Compiler Algorithms for Synchronization*, IEEE transactions on Computers, vol. C-36, No. 12, pp. 1485-1495, Dec. 1987.

[PCF88] *PCF Fortran: Language Definition*, The Parallel Computing Forum, August 1988.

[Scho89] *On-The-Fly Detection of Access Anomalies*, Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 285-297, June 1989.