AUTOMATIC DETECTION OF NONDETERMINACY IN PARALLEL PROGRAMS[*]

Perry A. Emrath
David A. Padua

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

1. Introduction

Many of today's computers, for example the Cray X-MP, Cray 2, ETA^{10}, Alliant FX/8, Sequent Symmetry, and Encore Multimax, are multiprocessors. Multiprocessing is a very appealing approach to parallelism since it can be used not only to accelerate the execution of a single program but also to increase throughput and reliability. Furthermore, the presence of independent control units makes possible the execution of parallel loops with branching, subroutine calls and random memory activity in a more effective way than is possible in single processor vector machines. However, writing and debugging a parallel program for a multiprocessor is, in general, more difficult than writing sequential vector programs.

A major reason for this difficulty is the need for explicit synchronization between components of a parallel program. There are ways to avoid having to use explicit synchronization, such as by using parallelizers or programming in functional languages. However, we believe that conventional parallel programming will remain an important approach in the foreseeable future. It is assumed here that data communication between concurrent components takes place through a (partially or wholly) shared memory.

A programmer may overlook the need for synchronization instructions, or may (by mistake) write them in the wrong order. A program with such errors in synchronization will most likely be *non-determinate*, that is, different runs (with the same input data) may compute different results.[1] Most likely, such a program would be incorrect, but the problem of debugging this kind of error is extremely difficult, and few tools, if any, are available for attacking such errors.

In some cases, non-determinacy may be intentional, for example, when the program implements an asynchronous algorithm [Kung76]. It is also clearly an integral aspect of operating systems, real-time systems, and process control. However, we believe many application programs, including parallel ones, are designed to be determinate. In these cases, non-determinacy is most often the result of a mistake on the part of the programmer, and these are the kinds of programs and problems on which we are focusing.

Our goal is to develop methods and tools for detecting non-determinacy in parallel programs at the source level. Part of the problem is to inform the programmer, in concise ways, of the cause of any non-determinacy found. Such tools are meant to facilitate the debugging chore by dividing it into two phases. If a programmer decides at the outset that his parallel program should be determinate by design, then tools to automatically detect non-determinacy can be used first in order to debug timing and synchronization errors. This phase would leave the programmer with a program that is ensured to be determinate, although not necessarily correct. The second phase might be to utilize an interactive break-point debugger to find, for example, arithmetic or logical errors. This can be done with assurance that timing differences will not affect results and that debugging sessions are repeatable.

[*]This work was supported in part by the National Science Foundation under Grant No. US NSF DCR84-10110, the US Department of Energy under Grant No. US DOE DE FG02-85ER25001, and a donation from the IBM Corporation.

[1] We will not consider deadlock here, though our approach can be used to detect this type of problem as well.

The next section of this paper discusses non-determinacy and provides classifications for different levels of non-determinacy. Examples of each class are shown. Clearly, the input data for a program may affect the determinacy of that program, and this issue will also be addressed.

The following sections describe the tools we are developing to detect non-determinacy. These tools involve two approaches. One is static and is based on analysis of the source program. The other approach uses an execution trace of the program being debugged. Trace analysis can be more thorough than static analysis, but it is generally slower and may be valid only for the specific input data used during the trace run.

Generally, when a programmer writes a sequential program for a uni-processor, the first debugging phase is to eliminate syntax/compiler errors. When the program compiles, one usually launches into executing it and then may use print statements, post-mortem traceback, or an interactive debugger to find arithmetic, logical, or control-flow errors. When the program is explicitly parallel for a multi-processor, the same techniques are woefully inadequate for finding errors due to non-determinacy. With the tools we are developing, one could first analyze the source code. This step will indicate statement pairs that can lead to non-determinacy, and also indicate whether it is conditional on the input data. From this point, the programmer might go back to edit the source program, or decide to employ the trace analysis tools.

Trace analysis works by first instrumenting the program to generate a trace of data accesses and recognizable synchronization points. The program is run with a given input data set to generate the trace. The trace is then analyzed and any non-determinacy that is found is reported. An important feature of the trace analyzer is that the trace run can be done sequentially (on a uni-processor) or in parallel. Either way, if the program is non-determinate for the given input, the trace analysis will verify this. If the tracing is done in parallel (on a multi-processor), multiple trace runs may produce different traces. However, obtaining different trace outputs is not necessary to demonstrate non-determinacy (though it is certainly a confirming phenomenon). If non-determinacy exists, trace analysis may find different causes from different trace runs, depending on the paths taken in each run.

The debugging tools described here are being implemented as part of the Faust project at the University of Illinois [GRMG88]. The scope of this project is to provide software development environments and tools for parallel programming. We are targeting our tools for use with Cedar Fortran, an extension of Fortran 8X that includes constructs for specifying parallel loops, concurrent tasks, and synchronization. Cedar Fortran is aimed at the architecture of the Cedar parallel processor [KDLS86], under development at the University of Illinois. The approach described here, however, is not necessarily restricted to Fortran dialects or the Cedar machine.

2. Non-Determinacy

In the field of computer science, the term *determinacy* usually refers to the property of repeatability. Likewise, a *non-determinate* system is one that exhibits non-repeatability, generating different results when started with exactly the same inputs. For our purposes, we are concerned with this notion of determinacy as applied to explicitly parallel programs. However, there is more to the problem of creating a *correct* parallel program than making it determinate. A determinate program may not be correct, and in some cases, a non-determinate program may be correct, being designated as such by the designer, implementor, tester, or user of the program.

When discussing the determinacy of a program, we may need to know something about the input data. A program may be determinate for some input data set, but non-determinate for a different input. It is easy to construct an example of such a program:

```
read *, I
if (I .NE. 0) then
      cobegin
            I = 1                    (1)
            I = 0
      coend
endif
print *, I
```

This program will always output 0 if its input is 0, but it may output 1 or 0 if its input is not 0. Such programs are probably not the norm, and we would like to automatically identify such behavior whenever possible. Due to problems such as these, the following discussion about classes of determinacy assumes the classification of a program holds only for a fixed set of input data. We will return to the question of different input data sets at the end of this section.

As programmers considering the task of developing parallel programs, we feel that it is useful to refine the notion of determinacy beyond that of a Boolean property. From a theoretical viewpoint, it is clear that a program, treated as a black box, is either determinate or non-determinate, possibly depending on the input data. Either the program always leads to the same results, or it can be demonstrated or proven that different results are possible. But from a practical point of view, we wish to further divide these cases. We consider a determinate program to be in one of two distinct classes, which we call *internally* and *externally* determinate. Non-determinate programs are divided into classes we call *associatively* and *completely* non-determinate. We'll discuss the significance of each of these classes later and provide examples to indicate the distinctions. But first, a discussion of our notion of threads and the cause of non-determinacies is needed.

For the purposes of this paper, each abstract sequential process in a parallel program is called a *thread*. Threads that exist at a given point in time may be in concurrent execution by different processors. The execution progress of each thread is unpredictable due to things such as clock tolerances, memory or network contention, interrupts, etc. During the lifetime of the execution of the program, many threads may begin and terminate. Many programming constructs have been devised over the years to implement similar notions of threads, such as `fork` and `join`, `cobegin` and `coend`, and more recently a variety of parallel loop constructs. For our purposes, when a particular thread causes other threads to begin, the original thread is considered to terminate at the point where it splits. The new multiple threads inherit all or some part of the original thread's address space. For example, sequential code prior to a parallel loop is considered a distinct thread from that of the sequential code following the loop, while each iteration of the loop is a separate thread. When a thread splits into multiple threads, or multiple threads terminate to form a single thread, the initial state of the new thread(s) is determined solely from the final state of the old thread(s).

In general, threads may share some (or all) variables, and may be able to use certain synchronization primitives which allow them to wait until some condition or event is satisfied. Some languages or architectures do not provide the abstraction of shared variables, but instead rely on the notion of message passing. However, we will not address the message passing paradigm, except to note that a receiving buffer in a thread behaves much like a shared variable that is to be written by many other threads. While the time at which a receiving buffer is filled may be controlled, the ordering of messages may not be determinate, just as when writing into a shared variable.

The cause of non-determinacy in a parallel program is the existence of races. A *race* exists when two statements in concurrent threads access the same variable in a conflicting way and there is no way to guarantee execution ordering. Accesses conflict if they are not both reads, that is, one statement is reading the variable and the other is writing it, or both are writing it. If a race occurs irrespective of the input data, it is called a *conclusive race* and causes a *conclusive non-determinacy*. If a race may occur for certain values of input but not for others, it is called an *input-dependent race* and leads to an *input-dependent non-determinacy*.

We call a program internally determinate if (and only if) the sequence of instructions each thread executes, along with the values of operands (variables) used by each instruction, is determinate. No races exist in such a program. This does not imply that the overall state of the program variable space is determinate. For example:[2]

```
dimension K(2)
K = 0
doall I=1,2                    (2)
      K(I) = I * I
enddo
```

is internally determinate, because the I-th thread of the loop will always change K(I) from 0 to I squared, but on some runs the array K may go through the sequence {(0,0), (1,0), (1,4)}, while on

[2] A `doall` loop is a concurrent loop. This means that the different iterations of the loop can all execute concurrently. Inside a `doall` loop the only form of synchronization allowed are those needed to control critical sections.

other runs K may go through the sequence {(0,0), (0,4), (1,4)}. The condition of determinate operands must hold even for variables that are shared with other threads. It follows that the sequence of values each (shared) elemental variable takes is determinate, and hence also the output of the program.

It should be clear that a program in which no two concurrent threads ever access the same variable is internally determinate. This is so because each thread is independent of others and so must be determinate. By induction from the first thread, the initial and final state of each thread must be determinate, and the conditions for internal determinacy are satisfied. Some parallel programming language constructs inherently enforce this, for example, the `fork()` system primitive in the Unix operating system[3] and the Burrough's `doall` loop construct [LuBa80]. But these "safe" constructs are often not flexible or efficient enough for the programmer of parallel numerical algorithms. In some cases, the compiler cannot always enforce the requirement of non–sharing between threads. Also, other constructs that can lead to non–determinacy may be available in the language. When actual sharing of variables does occur between threads, internal determinacy can still be achieved through the correct placement of appropriate synchronization instructions.

We call a program externally determinate if its output is determinate, but the program is not internally determinate. Races may exist, but the program will always produce the same output. Externally determinate programs generally arise when commutative and associative operations are performed on shared variables. Consider the following program:

```
read *, N
M = 0
doall I=1,N              (3)
      P(σ)
      M = M + I
      V(σ)
enddo
print *, M
```

where P and V are the well known synchronization primitives. This program will *always* output the sum of the first N integers, but there are N! possible sequences of values taken on by M. There are races between the read of M in one iteration and the write of M in another, and also between the writes of M by different iterations. But the P and V synchronizations guarantee that iterations do not interfere with each other. By the associativity and commutativity of integer addition, the final result is determinate. Programs that are externally determinate, as opposed to internally determinate ones, will be more difficult to debug using traditional (e.g. breakpoint) debugging techniques. However, many useful programs may be externally determinate, and tools to verify that a program is so rather than being non–determinate should prove very helpful.

One of the problems programmers must face when using floating point hardware is the fact that such machines only handle a limited precision. Care must be taken and careful analysis must often be done to ensure that results are not being corrupted by excessive round–off error or loss of precision. In floating point hardware, operators that are mathematically associative, such as addition and multiplication, are not associative for all possible values. This effect can lead to different results depending on the order in which the operations are carried out. Analysis can be performed to determine upper bounds for the amount of error that is possible in each output value [LaSa80].

In a parallel program, the order of floating point operations on a shared variable may vary from run to run, when done by different parallel threads. This can lead to non–determinate output. We call a program associatively non–determinate if the output is non–determinate because of only this effect. That is, the program would be externally determinate if floating point operations could be done with infinite precision, but in reality the output is non–determinate because of the non–associativity of the hardware floating point operations. The program

[3] Some recent incarnations of the Unix system provide the ability to mark certain memory to be shared between the parent and child processes of a fork(), allowing one to bypass this enforcement.

```
read *, N
X = 0
doall I=1,N
      P(σ)
      X = X + 1./I                    (4)
      V(σ)
enddo
print *, X
```

is an example of associative non-determinacy. Some parallel numerical programs will be associatively non-determinate, and yet will be considered correct programs. Generally, such programs should give very close to the same output on different runs, there being a bound on the amount of variance possible. The tools we are developing, possibly in conjunction with error analysis techniques [LaSa80], may prove useful in classifying and debugging parallel programs that exhibit behavior such as this.

Programs that don't satisfy any of the above mentioned conditions are called completely non-determinate. They have non-determinate output, and the range of the output values is not bounded by the nature of the operations performed, but by the timing of different operations on shared variables. It is easy to construct completely non-determinate programs, such as:

```
read *, N
doall I=1,N
      M = I                           (5)
enddo
print *, M
```

As shown in the introduction, the determinacy class a program belongs to may depend on the input data. For example, the completely non-determinate program above is only so for input values greater than one. For values less than or equal to one, it is internally determinate. An interesting case is (4) above, the associatively non-determinate example. Assuming the underlying hardware is binary and that 0.5 and 1.5 are represented exactly, this program is associatively non-determinate for input values greater than two, externally determinate for the input value two, and internally determinate for input values less than two. For the example given to demonstrate internal determinacy, this classification holds regardless of the input. The program given as externally determinate is in fact internally determinate for inputs less than two.

An ultimate goal is to be able to automatically determine if a program is in the same determinacy class regardless of its input data, or to identify ranges of input values for which a given program is in a particular class. The classes presented above seem to follow a natural progression from the least amount of non-determinacy to the greatest, in the order: internally determinate, externally determinate, associatively non-determinate, and completely non-determinate. Hence, in general, it seems reasonable to classify a program according to the least determinacy experienced for some input data set. That is, if a program is designated in some class without specifying anything about the input data, there is no input data set for which the program belongs in a less determinate class.

3. Source Program Analysis

One way to detect race conditions between statements is by analyzing the source code. This analysis may be performed by the compiler or by an independent program. In any case we will say that the analysis is performed at compile-time.

The source program analyzer will identify three types of situations. The first two are races that can be analyzed with precision at compile-time. These are the conclusive and input-dependent race situations mentioned above. The third type of situation is one where nothing can be said at compile-time due to limitations of the analysis algorithms. These are called *potential races*. The main purpose of the trace analysis phase described in the next section is to deal with potential races. However, it is conceivable that trace analysis could help to detect whether an input-dependent non-determinacy materializes for a particular input data set.

To illustrate the case of a potential race, consider the following segment of code

```
cobegin
      A = 1                           (6)
      A = 0
coend
```

where there is clearly a race between the two assignment statements. This can be detected at compile-time. On the other hand, assume the code is as follows:

```
subroutine A1(A)
      A = 1
end
subroutine A0(A)
      A = 0
end
. . .                                        (7)
cobegin
      call A1(A)
      call A0(A)
coend
```

If the program analysis routines lack the ability to do interprocedural analysis (i.e. it operates on a subroutine-at-a-time basis), then the identification of the race would not be possible at compile-time. The source program analyzer should in this case indicate that, in each subroutine above, the assignment to A may lead to non-determinacy depending on what is going on outside the subroutine.

An example of an input-dependent race is provided by the following segment of code:

```
    read *,K
    doall I=0,N
S:        A(K(I)) = ...                      (8)
    enddo
```

which may or may not be determinate depending on the values of K. The static analyzer can indicate that when K is not a permutation vector a race will exist.

In the previous two examples, the programmer can quickly analyze the segments of code and discover the potential problem. In more complicated cases tracing may be of help.

One of the functions of the source program analyzer is detecting associative non-determinacy and reporting its sources to the programmer. However, if the programmer asserts that associativity can be assumed safely, then no associative non-determinacies have to be reported. A similar situation arises in some of today's parallelizers where linear recurrences are vectorized only if the programmer indicates that associativity is to be assumed [Alli87].

Clearly, one of the goals in designing the source program analyzer is to decrease the set of potential races and therefore increase the set of races detected at compile-time. This is because the sources of non-determinacy identified at compile-time will not have to be traced or analyzed by the programmer, which will save time. Also, the information is more useful than that produced by the trace analyzer since this last only applies to the input data used when the trace was generated and does not give a clue as to whether this situation applies to all input data or a description of the input data that will generate races. One way to reduce the set of potential races is by increasing the accuracy of the algorithms. Another possibility is to accept *assertions from the programmer.* For example, the programmer may assert that in the previous doall loop, K is a permutation, and in this way guarantee that the loop will be determinate. One possible problem is that false assertions may be introduced. However, the compiler could be extended to optionally generate code to verify the assertions.

To detect races, the source program analyzer must detect two relationships. One is execution order and the other is the accessing of memory locations in a conflicting manner. We study these relations below. For the discussion that follows, we will call every execution of a statement a *statement instance.* The different instances of a statement will be identified by a superscript. For example, in the loop (8) above, the statement S has N instances, one for each iteration of the doall. When the different instances of a statement are generated by a counting (do) loop, we use the iteration number, not the index value, to identify the instances. Thus, the instances of S in the loop (8) are S^1, S^2, ..., and S^{N+1}. In the case of multiply nested loops, the superscript will be a list of integers, one for each enclosing loop.

Given two statement instances S^α and T^β, we say that S^α precedes T^β, denoted $S^\alpha<T^\beta$, if S^α is always guaranteed to execute before T^α. There are two ways by which the ordering may be guaranteed: by the flow of control of sequential threads or by synchronization instructions.

If S^α and T^β belong to the same thread, or one belongs to an ancestor of the thread to which the other belongs, then by analyzing the flow of control it can often be determined that either $S^\alpha<T^\beta$ or $T^\beta<S^\alpha$. Sometimes the execution order may not be determined. For example, consider the following segment of code.

```
x = .false.
cobegin
      x = .true.
      do I=1,N
            if x then go to T
      S:     ...
            if x then go to W        (9)
      T:     ...
            if x then go to S
      W:     ...
      end do
coend
```

In this loop, the order of execution of S^i and T^i will depend on the time at which the assigment to x is made. If both S^i and T^i, for some i in [1,N], access the same location in a conflicting way (i.e. both write or one reads and the other writes to the location), then we will have non–determinacy. But this would be a consequence of another non–determinacy caused by the (conclusive) races between the assignment to x and the reads that take place in the if statements.

Ordering can also be guaranteed by synchronization operations. Some of these operations can be easily detected and analyzed, especially when the synchronization is performed explicitly by invoking intrinsic functions. In general, detecting synchronization is more difficult and sometimes we believe impossible since busy waiting can be performed in many contorted ways. Undetected synchronization may lead to spurious warning messages. However, it is expected that the users of this debugging system will use explicit mechanisms for synchronization that are possible to analyze.

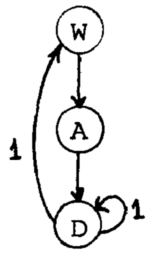
To analyze synchronization, the source program analyzer will build graphs, called *ordering graphs*, whose nodes represent statements, and whose arcs represent execution order between statement instances. The arcs will be annotated with information about the relationship between the statement instances whose order they represent. Consider for example the following loop.[4]

[4] A doacross loop [Padu79] is a concurrent loop where two types of synchronization constructs are allowed: synchronization to control critical sections and synchronization that goes from lower to higher iterations.

```
      doacross I=0,N
W:          await(E,I)
A:          A(I) = A(I-1) + 1          (10)
D:          advance(E,I)
      end doacross
```

The synchronization is performed by events (E is the name of an event) [ReKa79]. Each event is initialized to 0 when the doacross starts. The advance instruction first waits for the value of the event in its first parameter to become equal to its second parameter, and then increments the event by one. The await instruction waits until the event indicated in its first parameter is equal to or greater than the integer in its second parameter.

The ordering graph for the previous loop is

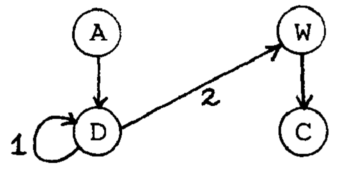


where the 1 on the arc from D to W specifies that W^{i+1} executes after D^i. The arc from D to itself is due to the waiting that takes place on the advance instruction. The arcs from W to A and from A to D have no annotation which means that A^i executes after W^i and D^i after A^i.

The annotation is easy to compute when both statements are nested inside the same loop, and it can be extended so that it can be applied in some cases when the statements are in different loops. Consider the following loops:

```
cobegin
      doall I=0,N
A:          A(I) = B(I) + 1
D:          advance(E,I)
      enddo
      doall J=0,N                      (11)
W:          await(E,J-1)
C:          C(J) = A(J-2) + 2
      enddo
coend
```

Here, the ordering graph has the form

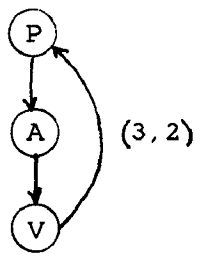


where the 2 in the arc from D to W means that instance W^{i+2} is executed after instance D^i.

The annotations shown in the previous two examples represent the *distance* between the statement instances being sequentialized by the synchronization statements. This distance may be an integer as in those two examples or could be a vector of integers with each element of the vector corresponding to a different loop nest as in the following example.

```
doacross I=4,N
      do J=3,N
P:          P(σ(I-3,J-2))
A:          A(I,J)=A(I-3,J-2)+1        (12)
V:          V(σ(I,J))
      enddo
enddo
```

Again, P and V are the synchronization primitives. The ordering graph for the previous loop is:



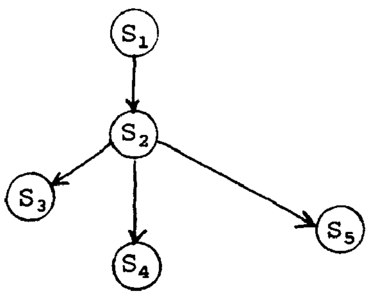
In some cases the distance may not be a constant or may not be possible to compute by the techniques used in the source analyzer. In these cases we will use a special mark as annotation.

The synchronization that forces order between statement instances takes place between two instructions: a *signal instruction* (such as V or advance), and a *wait instruction* (such as P or await). To compute the ordering graphs it is necessary to pair the synchronization instruction instances that will actually synchronize. The algorithm to perform this pairing will obviously depend on the type of synchronization statement.

When more than one signal instruction may fire a particular wait instruction, the actual ordering relationship exists (if the distances are identical) between any common ancestor of all the signal instructions and the wait instruction. For example, in the following segment of code

```
cobegin
      begin
            S1
            S2
            cobegin
                  begin
                  ...
            S3:   V(σ)
                  ...
                  end              (13)
                  begin
                  ...
            S4:   V(σ)
                  ...
                  end
            coend
      end
      begin
      ...
S5:   P(σ)
      ...
      end
coend
```

the order can only be guaranteed between S_2 and S_5, as shown in the following graph:



The second relationship between statement instances needed to detect races is whether they access the same memory location in a conflicting manner. In the absence of subscripts, name analysis suffices to detect conflicting access to memory. Well known dependence analysis techniques are needed in the presence of subscripts [Bane88], [PaWo86]. Thus, in the loop

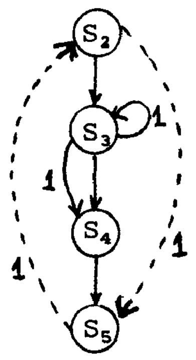
```
    doall I=1,N
S1:         A(I) = ...                    (14)
S2:         ... = A(I+1) ...
    enddo
```

we find by using data dependence analysis techniques that S_1^i and S_2^{i-1} access the same element of A in a conflicting way.

The *conflict relationship* will be represented in the ordering graph by using arcs annotated with a distance, when it is known. These arcs will be called *conflict arcs* and are distiguished from the arcs representing ordering by using dashed lines. The conflict arcs will point in the direction indicated by the distance. When the distance is unknown, the arcs should be annotated with a special mark as was done with the ordering relationship arcs. For example, in the loop

```
    doacross I=0,N
S2:         A(I)=B(I)+D(I-1)
S3:         advance(E,I)                  (15)
S4:         await(E,I)
S5:         D(I)=A(I-1)+B(I)
    enddo
```

the ordering graph has the form:

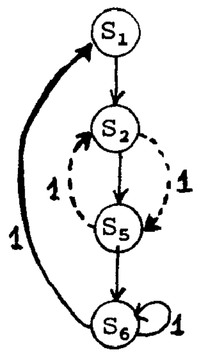


The source program analyzer uses the ordering graph to detect conclusive, input-dependent, and potential races. A conflict relationship will lead to a race if no order can be guaranteed between the statement instances involved in the conflict. To guarantee that a statement instance S^α executes before another statement instance, say T^β, it is necessary to find a path of ordering arcs from S to T of length $\beta-\alpha$. This length is formed by adding the annotations associated to the arcs in the path. Clearly, if the annotations are vectors of integers, the addition of the annotations has to be a vector addition.

In loop (15) there is a race as can be seen from the ordering graph. There is a conflict arc from S_5 to S_2 of length 1, but there is no path of ordering arcs from S_5 to S_2. However, in the loop:

```
    doacross I=0,N
S1:         await(E,I)
S2:         A(I)=B(I)+D(I-1)              (16)
S5:         D(I)=A(I-1)+B(I)
S6:         advance(E,I)
    enddo
```

there are no race conditions. To see this, let us analyze the ordering graph of this loop:



In this graph there is a path of ordering arcs of length 1 from S_2 to S_5,[5] and another from S_5 to S_2 which is a sub-path of the previous one. These two paths cover both conflict relationships, and this shows that there is no race. Unfortunately, the loop is now serial, but if there were other statements in the body of the loop we could still have parallelism, and be determinate.

[5] This path of length 1 is formed by the ordering arcs (S_2, S_5), (S_5, S_6), (S_6, S_1), (S_1, S_2), and (S_2, S_5).

The general rules to be applied are as follows: For a conclusive race to exist, the distance of the conflict must be known, and all the ordering paths (i.e. paths formed by ordering arcs) between the nodes must have a known length (i.e. no special marks since these mean unknown or variable distance) which is always different from the distance on the conflict arc. A conflict always gives rise to a conclusive race if no ordering path exists between the conflicting statements.

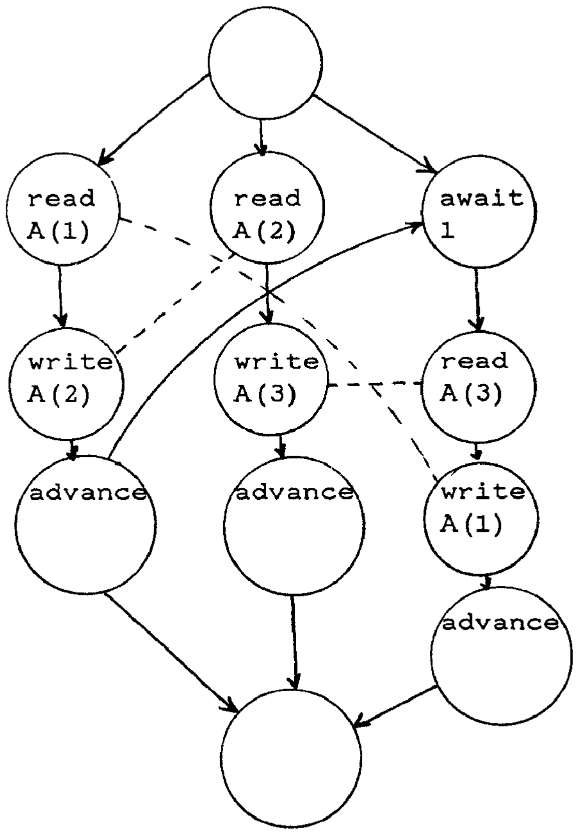
A race will be potential if either the conflict relationship is annotated by a special mark or no ordering path has a distance equal to the one on the conflict arc, and at least one of the ordering paths connecting the two conflicting statements contains a special mark.

4. Trace Analysis

Trace analysis could be applied when a potential race is discovered, that is, a race that cannot be verified statically. For example, consider the loop:

```
doacross I=1,N
      if (I .gt. 2)
         then await(E,I-2)
V1:                ... = A (K(I))      (17)
            A (J(I)) = ...
            advance(E,I-1)
enddo
```

Since the (potential) race between V_1 and V_2 cannot be confirmed statically, we could instrument the program in such a way that references to A in the doacross loop are traced. The trace includes the memory addresses being referenced, the type of reference (read or write), the synchronization operations, an indication of when parallel constructs start and finish execution, and several other entries. The instrumented program is executed and the trace is analyzed in much the same way as the static analyzer examined the source program. A graph is built from the trace, and analysis is performed on this graph. To illustrate this, let us assume that in (17), K = (1, 2, 3), and J = (2, 3, 1). The resulting trace can be transformed into the following graph:



In this graph, the dashed edges indicate conflicting access to the same variable, and the arcs indicate execution ordering constraints. In this example there are only two races. The read and the write of A(1) are *not* a race, since the two references are ordered by the advance and await instructions on E.

In some cases, trace analysis cannot verify potential races. Consider the program:

```
      doall i=1,N
W1:          P(i) =
W2:          if (P(K(i))) then
                 A(i) = A(i-1)
                 end if
      end doall
```

The potential races between instances of W_3 cannot be verified statically. Furthermore, since there is a race between W_1 and W_2, the particular execution used to generate the trace may not show a race between instances of W_3, even though one may exist. The trace analyzer will still indicate non-determinacy in the program due to the race between W_1 and W_2. In cases such as this, the program would have to be traced a second time after this race has been removed.

5. Conclusion

In this paper we have presented some preliminary ideas on the design of a software tool for the detection of non-determinacy in parallel programs. Non-determinacy is not necessarily an error in a parallel program. For example, programs implementing asynchronous algorithms are non-determinate by design. Also, non-determinacy arises when cooperating processes communicate through a bounded buffer, even though the program as a whole may be externally determinate.

We believe that successful detection of the sources of non-determinacy would be quite helpful since timing problems are hard to detect with a conventional debugger. Our ultimate goal is to be able to distinguish between an external non-determinacy that could be an error and an internal non-determinacy that does not affect the final outcome of the program. Both of these phenomena have to be reported to the user but they should be distinguished automatically whenever possible.

Also, a good user interface has to be designed. One aspect of the interface should facilitate the use of a data base to store and retrieve the information produced by the tools described here. This will make it possible for the user to select only that part of the information of interest to him at a given time. In this way the user may avoid being overwhelmed with redundant or unnecessary information.

A second part of the user interface should accept information from the user, either in the form of assertions or commands. The assertions will reduce the number of races that have to be reported. The commands could be used to restrict the tracing to certain areas of the program.

As we said in the introduction, we are implementing this tool as part of the Faust programming environment being developed at The Center for Supercomputing Research and Development at the University of Illinois. There is a preliminary implementation due to T. Allen, who also contributed important ideas to this project. This implementation is being enhanced since we need efficient algorithms to conduct experiments on the use of the proposed debugging system.

6. References

[Allia87] Alliant Computer Systems Corporation, *FX/FORTRAN Language Manual*, Littleton, MA, May 1987.

[AlPa87] Allen, T., and Padua, D.A. "Debugging Parallel FORTRAN on a Shared Memory Machine", *Proceedings of the 1987 International Conference on Parallel Processing*, The Pennsylvania State University Press, University Park, Pennsylvania, 1987.

[Bane88] Banerjee, U., *Dependence Analysis for Supercomputing,* forthcoming, Kluwer Academic Publishers, Boston, 1988.

[GRMG88] Guarna V. A., Malony A. D., and Reed, D. A., and Gannon, D. B. "The Design of a Parallel Programming Environment," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Report No. 770, 1988.

[KDLS86] Kuck, D. J., Davidson, E. S., Lawrie, D. H., and Sameh, A. H., "Parallel Supercomputing Today and the Cedar Approach", *Science 231,* 4740 (Feb. 28, 1986), 967-974.

[Kung76] Kung, H. T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," *Algorithms and Complexity,* Academic Press, New York, 1976, 153-200.

[LaSa80] Larson, J.L., and Sameh, AC.H., "Automatic Error Analysis of Numerical Algorithms — A Relative Error Approach," *Computing, Vol. 24, 1980, 275-297.*

[LuBa80] Lundstrom, S.F., and Barnes, G.H. "A Controllable MIMD Architecture", *Proceedings of the 1980 International Conference on Parallel Processing,* IEEE Press, New York, 1980, 19-27.

[Padu79] Padua, D.A., *Multiprocessors: Discussions of Some Theoretical and Practical Problems,* Rep. UIUCDCS-R-79-990, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[PaWo86] Padua, D.A., and Wolfe, M.J., "Advanced Compiler Optimizations for Supercomputers " *Comm. ACM 29,* 12 (Dec. 1986), 1184-1201.

[ReKa79] Reed, D.P., and Kanodia, R.K., "Synchronization with Eventcounters and Sequencers," *Comm. ACM 22,* 2 (Feb. 1979), 115-123.