Debugging Fortran on a Shared Memory Machine

Todd R. Allen and David A. Padua

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

Abstract

Debugging on a parallel processor is more difficult than debugging on a serial machine because errors in a parallel program may introduce nondeterminism. The approach to parallel debugging presented here attempts to reduce the problem of debugging on a parallel machine to that of debugging on a serial machine by automatically detecting nondeterminism.

1. Introduction

Parallel processors such as Cedar [GKLS83] and the NYU Ultracomputer [GGKM83], also known as the MIMD [Flyn72] or MES/MEA [Kuck82] class of machines, have great potential for exploiting parallelism, but their flexibility also makes them more difficult to program than single instruction stream machines. For multiple processors to be able to cooperate on a problem they require both communication and synchronization. Two models of communication are message passing and shared memory. At first glance, debugging in a message passing system appears easier than debugging in a shared memory system. In a message passing system communication and computation are separate and distinct. All communications must be explicitly stated with primitives denoting message send and/or message receive. In a shared memory system the target of an assignment statement can be thought of as a message send. A use of a variable can be thought of as a message receive. In a message passing system one knows when processors are communicating and which processors are communicating together while in a shared memory system every single shared memory reference is potentially a communication from any processor to any other or even all of the other processors. In a shared memory machine communications are a likely source of error and these communication errors are difficult to detect using conventional breakpoint debugging techniques. To detect a communication error using breakpoint debugging techniques one must be able to detect the error by examining a machine state that occurs shortly after the error. Because the error depends upon the order of execution of instructions in different instruction streams, it may occur infrequently, which makes capturing communication errors difficult. This paper presents methods which are being used in the design of a new type of debugging tool. It is assumed that once the possible sources of nondeterminism are identified the debugging process can proceed by using methods similar to those used in conventional debuggers. To detect nondeterminism, information gathered during both compilation and execution is used. Compile time analysis relies upon the methodology originally developed for dependence testing [Bane79] [Wolf82]. During execution a trace is made which is used to refine the compile time information.

This work was supported in part by the National Science Foundation under Grants No. US NSF DCR84-06916 and US NSF DCR84-10110, the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001, and by a donation from the IBM Corporation.

Throughout this paper, *code* refers to the instructions to be executed. *Input* is the set of values that is used by the code but not generated by it. This includes values obtained through I/O statements, values generated by OS services such as the time function, and even values of variables which are used before they are defined. A *program* is considered to be both code and input taken together as a unit. Two programs are the same only if both the code and the input are the same. Thus when speaking of multiple runs of a program it is implied that the input must be held fixed for a given piece of code.

A *deterministic* program is one in which the behavior of the program is always the same from one run to the next. A Fortran program executed sequentially is deterministic. The instructions of a serial Fortran program have a fixed meaning, and execute in a fixed order. If the behavior of the code changes, it must be due to changes in the input and by the above definition of a program we have different Fortran programs. A parallel Fortran program can be *nondeterministic*. Although the instructions of a parallel Fortran program are the same as those doing the computation in a serial Fortran program, the instructions of the parallel program execute on asynchronous processors and may have no order guaranteed. Consider the concurrent instruction streams of Figure 1.1 accessing the same locations in shared memory. In a *doall* loop each iteration of the loop is independent and can be executed as a separate process, which can proceed in any order. In Figure 1.1, assume that array *A* was initialized to zeroes before this loop. At the termination of this loop, each of the locations 2 through N of array *b* could either have the value i–1 (if the write of A[i] occurs before the read of A[i]) or the value 0 (if the read of A[i] occurs before the write of A[i]). If the program was deterministic up to this point, we now have $2^{(N-1)}$ different machine states possible at this point. The great explosion of machine states prohibits one from enumerating all possible paths a nondeterministic program can take.

```
        doall i=2,N
s1:       A[i] = i
s2:       b[i] = A[i-1]
        end doall
```

Figure 1.1. Nondeterminism of Concurrent Instruction Streams

2. Dependence–Driven Debugging

A common desire of people working with distributed systems is to be able to run a parallel program using some mechanism to determine the ordering of events as they execute. It is also important that the mechanism doesn't substantially change the execution ordering of those events [BFMS83]. This has also been called the probe effect [Gait85]. One mechanism suggested to provide information on the execution ordering is timestamping

DISCLAIMER

The following pages are an exact representation of what is in the original document folder.

[Lamp78]. Because every memory reference in a shared memory machine can be a communication, it is natural to want to treat each reference as an event. In this approach, one would have to record a timestamp from a clock to mark when each reference was made. Machine instructions don't execute at a fixed point in time; their execution spans many machine clock cycles. Depending on the connection scheme between processors and memory, it is even possible that memory requests to the same location will not be serviced in the same order that the requests were generated. Thus timestamps would need to be generated at the memory modules for each reference instead of being generated at the processors and associated with the instructions.

Instead of attempting to determine the order in which events occur, one can attempt to determine whether the effects of a given pair of events depends on order. Communications on a shared memory multiprocessor depend on order and these communications can be detected using the methods of data dependence analysis originally developed for restructuring compilers [KKLW80] [PaWo86]. Given two array references both within a set of nested loops, data dependence testing answers the question of whether the two array references are dependent, i.e. whether they will access any common locations, and if so, in what order do they access those common locations.

2.1. Terminology for Classifying Dependences

Restructuring compiler writers make a distinction between *flow, anti* and *output* dependences [PaWo86]. A flow dependence is the relationship between a definition and a use of a variable such that a value of the variable flows from the definition to the use. An anti-dependence is the relationship between a use and a definition of a variable such that the use precedes the definition which destroys the current value. An output dependence is the relationship between two definitions of a variable such that one definition of the variable supercedes the previous definition. To restructure a Fortran program for execution on a parallel machine, care must be taken to preserve the order of dependent references. When debugging it is useful to know the fact that communication exists between a given definition/use or definition/definition pair. If an order of the references isn't guaranteed the fact that there is communication will introduce nondeterminism. The word *dependence* implies an order. For example A depends on B implies B must occur before A can occur. Within a parallel program, if no order for a pair of references to the same location is forced then a *race* condition exists for the reference pair, and the relationship between the pair is called a *data race*. Consider the parallel loop of Figure 1.1. If one interprets the loop as a serial loop then the values would flow from an earlier iteration to the following iteration. But it is not a serial loop. Calling the pair of references to array A a "flow dependence" suggests that one wants data values to "flow" from one iteration to the next, and that the loop should be serialized to honor the dependence. It may be that the programmer actually desired the race with no preferred ordering of iterations. The assumption that a program would be corrected if all doall's with races are replaced by serial Do's is not correct. If the references come not from a loop, but from blocks of concurrent code, then there is nothing to indicate which if any of the possible serial orderings would be preferred. A debugger's purpose is to provide a programmer with a convenient means to study a program's behavior. Data races are a good place to focus one's attention. They are likely to be errors because of the nondeterminism they can introduce, but the determination of whether the synchronization is correct or not will be left up to the programmer.

The term 'read/write race' will be used to indicate unordered communication between a use/definition pair. Two unordered definitions of the same location will be called a 'write/write race'. Note that a race doesn't exist for two reads to a shared location since the result will be the same regardless of the order they occur. Read/write and write/write races are collectively referred to as data races. If synchronization in the parallel version preserves the ordering of a pair of references to the same memory location then it isn't a race, it is a dependence. If one is unable to tell whether a pair of unordered references access the same location then it will be termed a *potential race*.

Data races may or may not exist for the same code depending on the input given to the code. However, if a data race exists in a program it may be detected regardless of whether the program executes correctly or not for a given run. When called with the array A the bubble sort in Figure 2.1 will always work correctly regardless of the order that the iterations are performed in the doall loop. The reason that it will always work correctly is that due to the values in array A no iteration of the doall ever writes a location that another iteration also reads or writes. Using just the data of array A one is unable to detect by tracing the possibility of error through a data race because no data races can occur. The program could still execute correctly with the call on array B if the iterations occurred in the same order that they would if the loop was serial. However, with array B multiple iterations of the doall read and write the same locations and a data race exists. This can be detected regardless of whether the particular run executes correctly or not.

```
data N/4/, A/2,1,4,3/, B/4,2,1,3/

call sort(N,A)
call sort(N,B)

subroutine sort(N,X)
integer N, X[]
do i=1,N-1
  doall j=i+1,N
    if (X[j-1] > X[j]) call swap(X,j)
      endif
  end doall
end do
```

Figure 2.1. A Working? Parallel Bubble Sort

Some data races can be detected at compile time using the techniques of data dependence testing while taking into consideration the synchronization instructions. Static data dependence testing is conservative. The technique is to try a series of tests in an attempt to prove independence of a pair of references. If independence is never proven then dependence is assumed. This guarantees all dependences are found. One can also test in the opposite manner. Instead of attempting to prove independence one can attempt to prove that two references access the same location. If dependence testing is applied to an unordered pair of references, then a test which would indicate dependence for ordered references indicates the possibility of a race condition. Races discovered which can be proven in this way, will be called *static races*. An example of a static race is the pair of references to array A in Figure 1.1, if N can be statically determined to be greater than 2. The remaining races found, those assumed

because independence could not be proven, may or may not really exist. These will be referred to as *potential races*. An example of a potential race is the use of array X in Figure 2.1 and the call to subroutine swap which has access to array X. By tracing references to memory, one can determine if potential races really are races. If a particular location is actually read and written or multiply written from separate instruction streams between points of synchronisation a race is indicated. Races found at run time will be called *dynamic races* or simply data races. If the program in Figure 2.1 were traced, the call to sort with array B would produce both a write to X[2] in iteration j=2 of the doall and a read to this same location from iteration j=3, which would be an example of a dynamic race.

Using static analysis, one can learn a lot about the flow of data through Fortran code. To get exact information one must actually execute the code over a set of input data and record the results through tracing. Currently, data dependence testing can only be done practically if the subscript expressions are linear functions of do loop indices or linear induction variables. Also, accurate testing can only be done if the lower and upper bounds of the inductive sequences are known. What is needed is a way to refine the information generated at compile time with information gathered at run time.

3. Finding Data Races Through Tracing

Tracing is a powerful debugging mechanism. For a given set of data, one can get complete information on the behavior of a program by tracing everything that the program does. If the program is deterministic then a trace can be an exhaustive description of that program. Tracing of this nature is expensive. In addition to the resources required to generate and save a complete trace, the data generated can be very time consuming to analyse. It is possible to limit the amount of tracing in this system since we are concerned only with proving or disproving the existence of potential data races. To detect data races through tracing, memory references, task spawning instructions (doall, fork) and synchronisation instructions must be recorded as they are executed. A data race is indicated by a read and a write or two writes to the same memory location by different instruction streams where those instruction streams have no synchronisation ordering them.

The example of Figure 3.1 shows a hypothetical trace of a program. The instruction *test x* will wait until x is greater than the iteration number minus the dependence distance specified in the testset instruction. Figure 3.2 shows a structure representing synchronisation of statements in the sample trace. Downward arcs indicate statements that are synchronised by virtue of belonging to the same instruction stream. Horizontal arcs indicate statements that are synchronised through synchronisation statements. If a directed path exists between two statements, they are synchronised. If no directed path exists between two writes or a read and a write to the same variable then a race condition exists. These race conditions may then be presented to the user either graphically or textually. The references in the trace are grouped according to instruction streams. The number of processors used to generate the trace makes no difference, because the instruction streams are determined according to the statements which control the spawning and synchronisation of instruction streams and not according to what is executed on which processor. In this case we can see that with the testset distance d=2, iterations 1 and 2 (each of which is a separate instruction stream) have no synchronization and thus there are several race conditions between them. There are no races for d=1, even though $s4$ of each iteration is unsynchronized with the statements of following iterations, because no common locations are read and written by these statements.

```
integer A[3], B[4]
data A/1,2,3/, B/1,2,3,1/

    doall i=1,3
s1:   test (x)
s2:   A[B[i]] = A[B[i+1]]
s3:   testset (x) d
s4:   B[i] = sum(A, B[i])
    end doall

    function sum(vect, len)
    integer sum, len, vect[len]
    sum = 0
    do i=1,len
      sum = sum + vect[i]
    end do
    return
```

Sample Trace:

enter doall	iteration 2	iteration 3
iteration 1	s1: test x	s1: test x
s1: test x	s2: read B[3]	s2: read B[4]
s2: read B[2]	s2: read A[3]	s2: read A[1]
s2: read A[2]	s2: read B[2]	s2: read B[3]
s2: read B[1]	s2: write A[2]	s2: write A[3]
s2: write A[1]	s3: testset x d	s3: testset x d
s3: testset x d	s4: read B[2]	s4: read B[3]
s4: read B[1]	s4: read vect[1]	s4: read vect[1]
s4: read vect[1]	s4: read vect[2]	s4: read vect[2]
s4: write B[1]	s4: write B[2]	s4: read vect[3]
end iteration	end iteration	s4: write B[3]
		end iteration
		exit doall

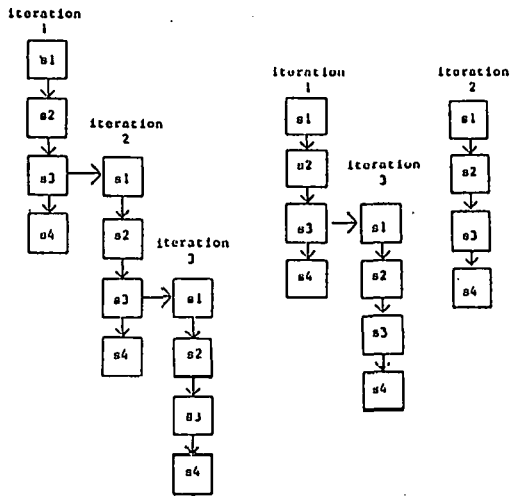
Figure 3.1. An Example of Tracing

3.1. Synchronization

To be able to detect nondeterminism in a parallel program, one must be able to recognize instructions which spawn multiple processes and the synchronization used needs to be understandable. Therefore, it is necessary to limit the primitives which will be accepted by the debugger. Synchronisation that is unrecognised by the debugger is not catastrophic. The debugger will simply report all reference pairs which may be in races without synchronisation. Unrecognised synchronization may increase the number of warnings but it will never cause races to go unnoticed.

3.2. Tracing Memory References

To record a memory reference, both the address referenced and the identification of the reference's lexical occurrence are needed. The address is needed to easily pick up conflicts that occur through aliasing. Actually, it is unnecessary to record the lexical occurrence of every variable referenced. A useful optimisation is to record basic block [AhUl77] numbers instead of individual references. At the entry point of each basic block, an integer that uniquely identifies that basic block will be recorded. Within the basic block there is a fixed number of references to be made, and they occur in a fixed order. Once a basic block is entered all of these references must be made. The only additional information needed to record the exact set of memory addresses referenced is the values of the indexing functions of arrays. By applying common subexpression elimination to the indexing func-



When testset distance d=1:
No race conditions.

When testset distance d=2:
Races.
s2: read of A[2] in iter. 1 and s2: write A[2] in iter. 2
s2: write of A[1] in iter. 1 and s2: read vect[1] in iter. 2
s2: read of B[2] in iter. 1 and s4: write B[2] in iter. 2

Figure 3.2. Analysing Sample Trace

tions, only the values of unique indexing functions need be recorded. Without recording basic block numbers one would have to trace scalars in addition to arrays. If the reference to a scalar was controlled by a conditional then the reference may or may not be in race with another lexical reference to the same scalar. However, by recording basic block numbers one has exact information concerning all scalars referenced. Recording basic block numbers to indicate what has been traced requires that the compiler build a table of basic blocks, where the entry for each basic block is a table of array references corresponding to the trace statements added for the indexing functions of those arrays. Recording basic block numbers moves work from run time to compile time. The compiler needs only to generate information of size proportional to the number of lexical references while one have to record information of size proportional to the number of dynamic references in the trace.

3.3. Analysing the Trace

One can do even better when tracing if some form of data compression is used. An example of this would be recording references in vector notation instead of on an element by element basis when it is obvious that an entire vector is going to referenced. This approach seems difficult to implement. When dealing with multi-dimensional arrays, one could record some subscripts in vector notation, but this may result in no savings at all if any subscript is nonlinear with respect to the innermost loop. In that case the nonlinear subscript would have to have its value recorded for every reference, generating just as much trace information as recording the value of the linearised indexing function for each reference. Even more important than saving during trace generation, using some form of vector notation also saves during trace analysis. Given two groups of references, if both groups of references are represented in vector syntax then the test for intersection is reduced to a single test for vector intersection. A reasonable compromise between the difficulty of handling the recording of references in vector notation and the benefits of doing so is to record all references on an element by element basis, and then construct vector notation before doing intersection testing. Constructing vector notation would be done by grouping all of the references generated by each lexical reference into vectors grown as large as possible during a single linear pass over the list of basic blocks executed. One can then test for intersection between two lexical references quickly by testing for intersection between the lists of vectors they referenced.

To actually detect races, a graph with two types of arcs will be built during static analysis. Each node of the race graph would represent a basic block and a scalar race arc would be placed between two nodes if a potential read/write or write/write race exists on any scalar if the two basic blocks were executed concurrently. The arc would be labeled with the lexical identities of all scalars involved in races between the basic blocks. Arcs are also created for array references. Array race arcs have the additional information of pointers to the locations of the array which are referenced. A race exists if both basic blocks representing the two ends of an arc appear in concurrent instruction streams and if the arc was for an array there must be an intersection between the locations referenced.

4. Using Dataflow Analysis to Direct Tracing

It is possible to guarantee that a program is free from data races through information gathered at run time. The program is deterministic until the first data race occurs thus, one is guaranteed of finding the first race. It will always occur if it exists. However, once the first race occurs, nondeterminism is introduced into the program and without further information one can no longer assume that any further results generated through tracing are accurate. This might not be a terrible limitation if the program being debugged does not require or is unable to take advantage of nondeterminism. Even so, this would still limit one to correcting data race errors one at a time. This is unacceptable since each trace can be expensive to generate.

4.1. The 'Hides' Relation

The difficulty of detecting multiple races at once lies in the fact that the races found introduce nondeterminism and a particular execution order may hide other races from the view of the tracer by preventing the occurrence of the race condition. For each data race there is a single variable or array element which ends up with a nondeterministic value. If the race is a read/write race then the nondeterministic variable is the variable which receives the value of the variable read. If the race is a write/write race, the variable involved in the race is the nondeterministic variable. If it is known where the nondeterministic variable is used, the effects of the nondeterminism can be isolated.

A test is needed that will indicate the possibility of one race hiding another race in the remainder of the trace. Although an exact test would be optimal, it is sufficient to be conservative. The test is conservative if it can guarantee that all data races are

found, but it may indicate races that do not exist. To do the static data race testing data flow analysis must be done. The same information can be used to determine the effects of the race. One race can hide a second race if it affects the locations referenced in the second race. If, for example, the second race involves an array, and if the nondeterministic variable of the first race was involved in either indexing function of the second race (either directly or through a chain of defs to uses) then the second race might not occur depending on the order of the references in the first race. Another way one race can hide a second race is if the nondeterministic variable of the first race appears in a control expression which can determine whether either reference of the second race is executed.

4.2. The Hides Graph

To deal with the problems discussed above we will use a graph whose purpose is to determine which of the potential races exist or must be assumed because tracing is unable to disprove a race condition. Let the static and potential races be the nodes of this graph. The arcs of this graph will represent the hides relation being true between two nodes. The nodes which represent actual races (races which are proven or cannot be disproven) will be marked and reported to the user. The relation *hides(A,B)* means that the occurrence of race $A \equiv \left(S_i, S_j\right)$ can effect the occurrence of a potential race $B \equiv \left(S_x, S_y\right)$ where S_n is a statement which is involved in the race. This relation is also computed in a conservative manner (i.e. if it can not be disproven the relation is assumed to be true).

Let the races be the nodes of a hides graph. For each node A which is a potential or static race and each node B which is a potential race, a directed arc is placed from A to B if *hides(A,B)* is true. All nodes representing static races will be marked. All nodes which are at the head of an arc represent races which could be hidden by the node (race) at the tail of the arc so they must also be marked since tracing cannot disprove the possibility of a race. The marks are iteratively propagated along all arcs until no additional nodes can be marked. The set of nodes marked through the above propagation need not be traced. After tracing the unmarked nodes, all nodes which were found to be races through tracing are marked. The marks must be propagated again. All marked nodes are assumed to be races and as previously stated, are reported to the user. The user will also be told why each race is being reported, i.e. whether the race was found statically, was assumed because it was hidden by a static race, it was found dynamically through tracing, or it was induced dynamically.

In Figure 4.1 there are only potential data races. Therefore, there actually is the possibility that this loop could run deterministically as a doall. At first glance, the potential race S4,S4 looks like a static race, but since the execution of this statement is conditional, no race condition may ever occur. Potential races S1,S2 and S2,S3 are not hidden. They need to be traced. If a race occurs in either case, every potential race must be assumed by propagation through the hides graph. If no races were detected, then the potential race s4,s4 would need to be traced. This trace could be performed at the same time as the previous trace or it could be traced on a different run. The potential race s4,s4 can be traced on a separate run because everything on which this race depends has been proven to be deterministic since no races were found on any of its incoming arcs. This process continues until the nature of each potential race in the hides graph has been determined. One simplifying assumption that could be improved is the propagation of a race across every arc of the hides graph. The arcs of the hides graph are built during

```
   doall i=1,n
S1:  A[f(i)] = B[i] + C[i]
S2:  C[i] = A[i]
S3:  if (C[g(i)] > n) then
S4:     E[i] = E[i-1]
S5:  B[E[i]] = B[i]
   end doall
```

Static Data races: none

Potential Data races: S1,S2 S1,S5 S2,S3 S4,S4 S5,S5
(actually these should be at the reference level and not at the statement level)

Hides Graph:

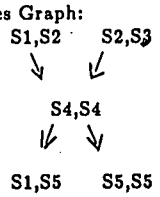


Figure 4.1. Sample Hides Graph

static analysis on a by name basis. If array C is nondeterministically assigned in S2, it is automatically assumed that this will effect the results of the test on C in S3. However, it may be that the one or more elements of C nondeterministically assigned are never actually used in the test in S3. In this case the race would have been unnecessarily propagated. Since tracing gives the specific sets of elements used, and the references responsible for the hides graph arc are either already being traced or could be added to the list of references to trace, it is possible that arcs created on a by name basis could be validated by checking the actual elements referenced. However, if races are unnecessarily propagated because the arcs aren't validated the user will receive excessive warning messages, but no races will be overlooked. Also, one would generally try and eliminate the root races before attempting to deal with the propagated races. If this is done, the falsely propagated races will automatically fall out if the root race is eliminated.

There is another case where one can get more accurate results. In Figure 4.2 a race on A can prevent execution of the references to B for some iterations of i. This causes an arc to be inserted in the hides graph since it cannot be determined whether the race on B exists by tracing. If all references made to B in both the false and true branches of the *if* are recorded regardless of which branch is executed then all locations of B that could be referenced are enumerated. By using this method one can detect whether a real race on B could ever exist.

If static analysis of race conditions is good, much work can be eliminated for the tracer. However, even if the static analysis is suboptimal the final results from the trace analysis will not be debased. Given two potential races A and B if the compiler proves no race on B, then the trace analyser will show no race on B occurred. The only way to assume a race on B after doing that trace analysis would be if there was a race on A and *hides(A,B)* were true. If *hides(A,B)* is true, then either the nondeterministic variable from A was used in the subscripting of B, and the compiler would have been unable to prove no race for B, or the nondeterministic variable from A affected the control flow around B, in which case if "conditional hides graph arc testing" was performed and showed a race then static analysis must also show a race.

```
doall i=1,n
A[f(i)] =
if A[g(i)] then
  B[h(i)] =
  endif
   = B[2i]
end doall
```

Figure 4.2. Testing of Conditional Hides Graph Arcs

Still there are several reasons to strive for accurate static analysis. First, static analysis can significantly reduce the amount of tracing needed. The more accurate the analysis, the less one needs to trace. Second, the hides graph arcs are all built upon statically generated data flow information. If the data flow analysis is poor, the hides graph arcs will be excessively conservative. Finally, results generated by static analysis are more general than the results of tracing, because static analysis applies to all possible inputs while the trace only applies to the inputs of the program traced.

Although trace results apply only to a single set of input data, if one were to include variables which are set through read statements as nodes in the hides graph, one could then determine if the occurrence of a race is affected by a change in input data.

5. Implementation

As stated in the introduction, the primary goal of this tool is to provide the user with knowledge about the nondeterminism in his program to allow him to apply breakpoint debugging techniques. This is accomplished in three phases: compile time static analysis, runtime tracing and trace analysis. Since little user interaction is needed for any of these phases and each phase could potentially be very time consuming for a large program, a noninteractive user interface is desirable. A prototype debugger using these methods is being designed as a batch system. The user will submit a job and wait for a listing containing information on the races. If this information is acceptable, the programmer will be able to proceed to an interactive debugging process. The interactive debugger will support breakpoints. The information generated by this tool will allow the user to make judicious use of breakpoints.

It was briefly mentioned before that the debugger will only be able to support a limited set of synchronization and task spawning primitives. The prototype debugger will accept the following statements. For the purpose of splitting a single instruction stream into multiple threads of execution, a forking subroutine call will be used. Synchronization between forked tasks is currently under consideration. For executing the iterations of a loop concurrently, the notation *doall* is used. The instructions *test* and *testset* [MiPa86] are used to guarantee ordering between references in concurrent loop iterations.

6. Merging These Methods with Breakpoint Debugging

Debugging is the process used to determine why the output of a program isn't what was expected. Serial debugging relies heavily upon breakpoints. A breakpoint interrupts the execution of a program and gives control to either a user who will examine/modify the program state or a debugging routine which will save some portion of the program state in a trace file. If a program is found to be deterministic or is altered to be deterministic then one can proceed with standard breakpoint debugging techniques. If necessary one can even sequentialise the execution by simulating a multiprocessor with a single processor. However, if the program retains some nondeterminism due to parallelism one must decide if breakpoint debugging is still applicable. Before one can evaluate the usefulness of breakpoints in a parallel program one must decide what a parallel breakpoint does. In a parallel program there are two possible implementations of a breakpoint, a local breakpoint that interrupts only the processor encountering it and a global breakpoint that brings all processors to a halt. Of these two choices the global breakpoint seems to be both more powerful and more difficult to implement. One question immediately surfaces with global breakpoints. Does it have to stop all processors immediately or can one allow some amount of delay between the time first processor hits the breakpoint and the time the other processors are halted? If all processors must be stopped immediately either the debugger will require special hardware support or the machine will have to be emulated. If the processors aren't stopped immediately they will have an opportunity to modify the state, perhaps changing the information in which one is interested. If a local breakpoint is hit by one processor each of the other processors will proceed until they reach a breakpoint, a blocking synchronisation or the ends of their instruction streams. With an instantaneous global breakpoint on an asynchronous machine each processor could be at any given point of progress in the execution of its instruction stream when the breakpoint is hit. The ability to stop the machine gives the illusion that one can pick a point to stop the machine and see what is going on. However, one is actually only picking the stopping point for the processor encountering the breakpoint. Trying to make deductions based on the current machine state is dangerous, if that machine state was randomly selected and the user has no information on how that state was reached. Local breakpoints give the user precise control. With local breakpoints the user can independently choose where to stop each processor. By using race analysis to determine where nondeterminism is introduced, the user can then place local breakpoints before each of the references involved in a race and determine what the effects of the race will be.

7. Conclusions

The purpose of a debugger is to aid a programmer in locating code which causes unexpected behavior in an application. In a parallel program, code which can introduce nondeterminism is likely to introduce unexpected behavior. By providing a mechanism to automatically detect nondeterminism we hope to aid the programmer in determining whether a problem is due to incorrectly specified parallelism or a logical error. After the sources of nondeterminism have been identified, the user will be able to proceed with breakpoint debugging techniques.

8. Acknowledgements

We would like to express our thanks to Sam Midkiff of CSRD, who inspired using data dependence information for debugging communication problems. Thanks are also due to Ron Cytron of IBM, who pointed out the usefulness of dataflow analysis. Dennis Gannon and William Jalby of CSRD provided, from their own practical experience, many insights on debugging. Thanks also go to Williams Harrison for help in the preparation of this paper.

REFERENCES

[AhU177] A.V. Aho, J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977

[Bane79] U. Banerjee, "Speedup of Ordinary Programs", Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-989, October 1979

[BFMS83] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, G. Vaglini, "Development of a Debugger for a Concurrent Language", *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on High-Level Debugging*, pp 98-106, Aug. 1983

[BuCy86] M. Burke, R.G. Cytron, "Interprocedural Dependence Analysis and Parallelization," *Proceedings of the Sigplan Symposium on Compiler Construction*, 1986

[Cytr84] R.G. Cytron, "Compile-time Scheduling and Optimisation for Asynchronous Machines", Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-84-1177, 1984

[Davi81] J.R. Beckman-Davies, "Parallel Loop Constructs for Multiprocessors," M.S. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-81-1070, May 1981

[Flyn72] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers*, Vol. C-21, No. 9, pp. 948-960, Sept. 1972

[Gait85] Jason Gait, "A Debugger for Concurrent Programs," Software-Practice and Experience, 15, 539-554, June 1985

[GGKM83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — Designing an MIMD Shared-Memory Parallel Machine," *IEEE Trans. on Computers*, Vol. C-32, No. 2, pp. 175-189, Feb. 1983

[GKLS83] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "CEDAR - A Large Scale Multiprocessor", *Proc. of International Conference on Parallel Processing*, 524-529, Aug. 1983

[Kuck78] David J. Kuck, *The Structure of Computers and Computations Vol I.*, John Wiley & Sons, 1978

[Kuck82] David J. Kuck, "High Speed Machines and Their Compilers," *Parallel Processing Systems*, pp. 193-214, 1982

[KKLW80] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectoriser for Pipelined Processors," *Fourth International Computer Software and Applications Conference*, Oct. 1980

[Lamp78] L. Lamport, "Time, Clocks and the Ordering of Events in Distributed Systems", *Comm. of the ACM*, Vol. 21, no. 7, pp. 558-565, July 1978

[MiPa86] S.P. Midkiff, D.A. Padua 'Compiler Generated Synchronisation For Do Loops', *Proceedings of the 1986 International Conference on Parallel Processing*, pp 544-551, Aug. 1986

[Myer79] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979

[Padu79] D.A. Padua, "Multiprocessors: Discussions of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-990, Nov. 1979

[PaWo86] D.A. Padua, M.J. Wolfe, "Advanced Compiler Optimisations for Supercomputers," *Communications of the ACM* Vol. 29, No. 12, pp. 1184-1201, Dec. 1986

[Smit85] Edward T. Smith, 'A Debugger for Message-based Processes', *Software-Practice and Experience*, vol. 15, 1073-1086, Nov. 1985

[Wolf82] Michael J. Wolfe, "Optimising Supercompilers for Supercomputers", Ph.D. Thesis, University of Illinois, Report No. UIUCDCS-R-82-1105, Oct. 1982