Dependence Analysis for Outer-Loop Parallelization of Existing Fortran 77 Programs

Josef Stein
Geoffrey C. Fox

CRPC-TR92261
July 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Submitted to Concurrency: Practice and Experience

SCCS - 339

Dependence Analysis for Outer-Loop Parallelization of Existing Fortran77 Programs

by
Josef Stein and Geoffrey C. Fox

Technical Paper

submitted to
Concurrency: Practice and Experience

July 1992

Syracuse Center for Computational Science
Syracuse University
111 College Place
Syracuse, New York 13244-4100
<sccs@npac.syr.edu>
(315) 443-1723

SCCS–339

Dependence Analysis for Outer-Loop Parallelization of Existing Fortran-77 Programs

J. Stein and Geoffrey C. Fox

Syracuse University

Northeast Parallel Architectures Center

111 College Place

Syracuse, New York 13244-4100

yossi@shum.huji.ac.il

gcf@nova.npac.syr.edu

Dependence Analysis for Outer-Loop Parallelization of Existing Fortran-77 Programs

JOSSI STEIN
GEOFFREY C. FOX
Syracuse Center for Computational Science
Syracuse University
111 College Place
Syracuse, New York 13244-4100
yossi@shum.huji.ac.il, gcf@nova.npac.syr.edu

1. INTRODUCTION

Many of the existing Fortran programs are, in principle, serial representations of parallel algorithms. We distinguish between inner-loop or fine-grain parallelism, which is usually represented by short do-loops, and outer-loop parallelism, which is usually represented by loops containing subprogram calls and serial inner loops. From the data point of view, in an inner-loop parallelism each micro-process deals with scalars or single elements of arrays, while in the outer-loop parallelism each microprocess may deal with large sections of arrays. Therefore, if a program was not originally intended to run on a parallel computer, its inner loops have a good chance of being parallel, while the outer loops will seldom reflect the parallel nature of the algorithm. Consequently, outer-loop parallelization of existing real-life Fortran-77 programs may become a formidable task. It is made even more difficult by the lack of appropriate debugging tools for parallel programs. Parallelism may be present at all levels of loop hierarchy. One must look for both inner and outer loop parallelism to exploit full possibility inherent in a problem. "Outer loops" are defined operationally as those containing significant internal structure, such as subroutine calls, which tend to make the job of an automatic parallelizer hard or even impossible. Note that an "outer loop" is not necessarily "outermost loop".

From the point of view of a user, who had already made up his mind to parallelize his program, there are three major problems:

1. How fast will the program run on a given parallel computer?
2. How much help can one expect from parallelization tools?
3. How portable will the parallelized program be?

In this discussion we address the last two problems, restricting our attention to loosely-synchronous programs [1]. The target machines are MIMD parallel computers, with shared OR DISTRIBUTED memories.

Our goal is to present some shortcomings of the static dependence analysis performed by current parallelization tools, and suggest

(a) enhancements to the static dependence analysis, and

(b) the use of dynamic dependence analysis.

In Section 2 we discuss features of a loosely-synchronous program. In Section 3 we discuss outer-loop parallelization of loosely-synchronous programs, and the use of dynamic analysis tools for detecting parallelization errors. In Section 4 we discuss results of an experimental analysis of some "real life" programs, using several parallelization tools, and in Section 5 we draw some conclusions from the experiments. Appendices I and II describe features of a source-to-source compiler for dynamic dependence analysis. Appendix III contains short descriptions of the programs which had been used in our experiments, and Appendix IV presents some additional loops which could not be parallelized by the tools which we had used.

All the codes and code-fragments mentioned in this work are available electronically from NPAC.

2. LOOSELY SYNCHRONOUS PROGRAMS

A loosely-synchronous program can be divided into sections of three types: Parallel, sequential and communication, with synchronization at the end of each parallel and communication section. Each instance of a serial or parallel loop may be subdivided into sections of the three types defined above.

- The parallel sections may contain neither communication nor synchronization.
- The communication sections are explicitly deterministic. The value of a variable may change only once in a communication section, and if it is changed—it cannot be used in the same section. Here we consider also Input/Output as a form of communication.

Communication between parallel processes differs from one computer to another. In order to keep the discussion independent of any hardware peculiarities we view a loosely-synchronous program as a set of communicating processes, considering sharing of data as a special type of communication. In this picture we assume that the sequential parts of the program are executed in each of the parallel processors.

Loosely synchronous programs have several advantages over general parallel programs:

2.1 Simplicity

The greatest advantage is its simple structure of three types of sections. This means less bugs to begin with, and relatively simple corrections to bugs, when they are found.

2.2 Portability

In general, parallel programs comprise computation and communication/synchronization. Assuming that the computational part is portable, because it is written in a portable version of Fortran, then the portability of the whole program depends on the portability of the communication/synchronization parts. A loosely synchronous program contains no explicit synchronization, and the communications can be written in the machine-independent form, by using COMMUNICATION PRIMITIVES like:

$$A(I) \leq B(J(I)),$$

where I and $J(I)$ represent sets of indices (null for a scalar), and A and B are separate, partitioned arrays. The portability will finally depend on a rich communication library. In many cases several simple communication primitives (like those provided in EXPRESS OR COMMERCIAL SYSTEMS SUPPLIED WITH INTEL NCUBE OR TMC multicomputers) are sufficient.

Porting parallel programs to/from shared-memory machines depends, of course, on the compilers. We assume that a compiler on a shared-memory machine recognizes shared variables and private veriables of all types.

Porting a loosely-synchronous parallel program from a message-passing to a shared-memory machine is trivial, in the sense that we can leave all variables local, and simulate the communication primitives.

Porting a parallel program from a shared-memory to a message-passing machine is also simple, when the program is loosely-synchronous. When a loosely-synchronous program is parallelized for a shared-memory machine, then its shared variables cannot be changed in the the sequential parts, because there is no synchronization there. They can be changed in the parallel loops, but only in a way which insures no loop-carried dependences. Therefore, the actual sharing of data between the parallel processes occurs in the communication parts, and at the implicit synchronization between the communication and the parallel sections. In order to make the program portable, one has simply to declare which variables are communicated at each parallel section. (An ideal compiler would find the information by itself.)

In this discussion we ignored the efficiency issues of the portability. Making a ported parallel program run efficiently on the new machine will usually require some modifications, like partitioning of arrays, or rearranging the order of the communications.

2.3 Dynamic Dependence Analysis of Parallel Programs on Serial Computers

A loosely synchronous form imposes restrictions on the programs. Programs which had been designed in a loosely-synchronous form, with these restrictions in mind, should, in general, be easy to debug. However, SERIAL programs are not subject to those restrictions. So, when, in the process of parallelization of a serial program, a loop is declared as being parallel—it may still contain dependences which inhibit its parallel execution. These dependences may be difficult to find, especially in a large program. One way of detecting such dependences would have been by using STATIC DEPENDENCE ANALYSIS, whenever it is feasible. Our study, which is described in the next chapter, suggests that current tools are still limited in their applicability to the analysis of complicated loops. The practical substitute for the unattainable verification is DEBUGGING. Almost all the existing numerical codes have been debugged, but not verified. Debugging of a parallelized serial program on a MIMD machine is considered to be far more difficult than debugging of the original serial program on a serial machine. However, this task can be greatly simplified with the help of a DYNAMIC DEPENDENCE ANALYSIS tool, which allows for the debugging of a LOOSELY-SYNCHRONOUS PARALLEL PROGRAM on a SERIAL COMPUTER.

a. Debuggability of Parallel Sections on a Serial Computer

   A parallel section may contain neither communication nor synchronization, therefore, a correct parallel section does not contain any data dependences between parts which should run in parallel, like e.g. loop-carried dependences. This property, while difficult to prove, can be easily analyzed dynamically on a SERIAL computer, by using a simple source-to-source compiler. (We consider a compiler to be "simple" if it does not include any dependence analysis. The reason for this will become apparent in the following sections. However, limited dependence analysis could improve the efficiency without really sacrificing simplicity).

b. Debuggability of the Communications on a Serial Computer

   All parallel programs need communication to pass information between processors, or between a processor and the outer world. Serial debugging of the whole parallel program, including the communication sections, requires a partitioning of data and of all the parallel-loop cycles among virtual parallel processors. The user has to partition the parallel arrays and loop-cycles, by using PARTITIONING PRIMITIVES [2], then the whole parallel program can be debugged on a SERIAL computer with the help of a second "simple" source-to-source compiler. The portable COMMUNICATION PRIMITIVES can be translated by the compiler into calls to reliable library functions.

NOTE that the restriction on communication and synchronization inside parallel loops is limited to the PROGRAM TEXT only. Communication and synchronization can be introduced by the compiler as a part of the *OPTIMIZATION*. This, however, should be transparent to the user.

3. OUTER-LOOP PARALLELIZATION

An "outer-loop" is characterized by its complicated body, which may contain loops and subprogram calls, including other "outer-loops". In the current discussion we ignore the subdivision of parallel outer-loops into Parallel, Communication and Serial sections. Subdivision can be added after the first division had been completed. Then steps 1–6 are performed to parallelize the inner loops.

We can define an outer-loop parallelization procedure which is divided into six steps:

1. Serial-to-serial transformation.
   a. Identification of the sections which represent the parallel algorithm.
   b. Changing the program in such a way as to make the parallelism explicit, but leaving the program in a sequential form.
   c. Serial debugging of the changed program.
2. Transforming the program into an explicit loosely-synchronous form, by removing all the parallelization-inhibiting dependences from the parallel sections.

   This is the most difficult step, and requires STATIC dependence analysis to find parallelization-inhibiting dependences in the potentially-parallel sections. At this step some communication might already be necessary in order to remove non-local references. In this case the communication primitives should be replaced (preferably with the help of a tool) by assignment statements, or library functions, and the program, which is still serial, though its potential parallelism is more apparent, can be debugged until it is as reliable SEQUENTIALLY as the original program.

This process can be demonstrated by the following example: Assume that the following program deals with N cells.

```
PARAMETER(N=10000)
DIMENSION  A(N),B(N)
.
.
DO   I=2, N-1
B(I)=FUNCTION_OF(A(I-1), A(I), A(I+1))
END DO
```

Arrays A(N) and B(N) represent some physical quantities in the cells. We intend to parallelize the program by dividing the cells among the available processors. Clearly, this loop contains a non-local reference to A(I-1) and A(I+1). On a shared-memory computer this could be solved by making A a shared variable, but such a solution won't be portable to other machines. A simple way to remove the dependence would be by using, for example, a CMFORTRAN/FORTRAN90—like SHIFT subroutine.

SHIFT(From, To, First, Last, Shift), which, on a serial machine, is equivalent to the loop:

```
DO   I=First, Last
To(I)=From(I-Shift)
END DO
```

The new program will then be:

```
DIMENSION A(N), B(N), Aleft(N), Aright(N)
.
```

```
.
CALL  SHIFT(A, Aleft, 2, N-1, -1)
CALL  SHIFT(A, Aright, 2, N-1, 1)
DO   I=2, N-1
B(I)=FUNCTION_OF(Aleft(I), A(I), Aright(I))
END DO
```

Now the non-local reference has been removed from the loop, and the communication-primitive SHIFT is used without any specific partitioning.

3. Dynamic dependence analysis of the parallel sections on a serial machine. This step comprises testing for parallelization-inhibiting dependences in the parallel sections, with the aid of a source-to-source compiler [see Appendix I] which adds machine-instructions for maintaining status information of the variables. For this purpose the user explicitly declares the parallel sections as parallel, by using constructs like PARALLEL DO, FORALL, etc.

   Note, that dynamic dependence analysis will, most probably, be necessary also to supplement the static analysis, by detecting violations of assertions, which cannot be verified statically.

4. Partitioning and communication.

   In this step the program is changed again. First a set of (virtual) processors has to be declared. Then the data and the parallel-loop instances are partitioned among the virtual processors and communication/synchronization is added where necessary. This step requires dependence analysis BETWEEN the loops, to find all the necessary communications.

5. Dynamic dependence analysis of the full parallel program on a serial machine.

   In this step the whole program is debugged on a serial computer with the aid of a source-to-source compiler which adds instructions for maintaining status information of all the variables [see Appendix II].

6. Parallel debugging on a parallel computer.

   This is the only step which is really machine-dependent.

   The process includes extensive and difficult dependence analysis, and, possibly, a fair amount of transformations on the source program. Therefore, the need for a good tool to help in the process is apparent. Ideally such a tool should replace steps 2-5 of the parallelization process, by performing both a full static analysis and the source-to-source transformations. However, experience with sequential programs has shown that even modest tools, like static detection of variables which are never used or never defined, can save a lot of debugging time. A tool which can STATICALLY detect parallelization-inhibiting dependences in a parallel program might be a great help to the programmer.

4. STATIC ANALYSIS TOOLS

In the last several years some very serious attempts have been made to construct static analysis tools. In fact the newly constructed tools are more ambitious than that: Not only are they supposed to find parallelization errors, but also help the programmer to correct them, by suggesting/performing various transformations on the source code. Some tools even try to find the parallel sections without using explicit parallel constructs, but this feature, though useful for fine-grain parallelization, does not seem to be an advantage to outer-loop or functional parallelization, because, almost always, the important sections become really parallel only after the tool has been used with extensive help from the user.

Static analysis tools for parallelization of Fortran-77 programs rely heavily on dependence analyzers. Results of the analysis are used in various ways. For the current discussion we are interested in the dependence-analyzer as a tool for finding parallelization- inhibiting dependences in parallel loops. (Step 2 of the parallelization process in Section 3).

Analyzers tend to be conservative, i.e. an analyzer declares a dependence whenever it cannot prove that there is no dependence. Since dependence analysis is almost never ideal, all the analyzers find, besides the real dependences, also spurious ones. Consequently, the better is the analyzer the less dependences it will find.

The current state of the art is that when analyzing a real (nontrivial) program one finds far too many spurious dependences.

We have spent some time trying to parallelize a small set of programs [see Appendix III] using the following tools:

- E/SP [3], PAT [4], FATCAT [5], PARASCOPE/PED [6], PFC/PSERVE [6] and PARAFRASE-2 [7].

The tools differ greatly in style and analysis power, but, since all of them are in various stages of development and debugging, we will not present a detailed comparison here. Of the 4 programs tested, only the simplest one, HYD, could be usefully analyzed by some of the tools. When analyzing the other programs, the tools reported too many spurious dependences, with too little information about each of them, to be useful.

A more detailed study of the performance of several of the tools, has shown that the dependence analysis may be improved in two different ways:

1. A more powerful propagation of symbolic values and status-attributes of variables and array-sections.
2. A better utilization of the user's knowledge.

4.1 Propagation of Values and Status Attributes

We demonstrate the need for the first improvement in a few simple, but typical examples. Note that "outer-loops" appear mainly in non-trivial programs, which are too large to be included in a paper. The following examples represent small pieces of some of the programs mentioned in Appendix III.

EXAMPLE 1:

Local arrays or sections of arrays are essential parts of outer-loops. Moreover, the boundaries of the array-sections are, in many cases (including all the programs I know) symbolic. In the following example one parallel loop (`DO I`) contains two sequential inner-loops: (`DO J` and `DO K`). The first one DEFINES a segment $A(1:N)$ of an array and the second one USES that segnent. Analysis of this program requires both a simple propagation of the symbolic value of the variable N (its numerical value is unknown to the analyzer) and the status of an array-segment $A(1:N)$ [8]. The propagation of the value of N is needed in order to find out that in the `DO J` loop it has the same symbolic value as in the `DO K` loop.

```
DIMENSION A(100),B(10)
READ *,N
IF(N.GT.100.OR.N.LT.1) STOP 'BAD N'
C   PARALLEL LOOP
DO I=1,10
C                            DEF LOOP
DO J=1,N
A(J)=J+I
```

```
END DO
C ------------------------------
S=0.0
C                              USE LOOP
DO K=1,N
S=S+A(K)**I
END DO
C ------------------------------
B(I)=S
END DO
PRINT *,B
END
```

It is obvious to the observer that the array A should be local to each of the parallel instances of the `DO I` loop, and that $A(1:N)$ is always defined before being used. (Some systems do not allow local arrays, but once a local array has been identified, it can be automatically transformed into an acceptable form by spreading.)

Nevertheless none of the tools recognized the parallelism of the `DO I` loop.

Two similar, but more complicated, examples are given in Appendix IV.

EXAMPLE 2:

Symbolic propagation is needed also for single-element dependence analysis: The following example includes three loops. Each of the loops changes the values of an array, element by element, by calling a subprogram to do the assignment, and then uses this new value. Each of the subprograms contains an `IF` statement, and changes the array element on both of its branches. The dependence analyzer must recognize the fact that each of the array-elements in each of the three loops is DEFINED BEFORE BEING USED. This analysis requires interprocedural, control dependent, propagation of symbolic values.

The first loop, DO I, passes the loop-index as a parameter to the subprogram SUB1 and the array A is shared through a COMMON block. The second loop, DO J, passes both the loop-index and the array, B, as parameters to the subprogram SUB2. In both the first and the second loops the loop-index has been used by the subroutine as a pointer into the array. (It is also used in the IF statement, but this is irrelevant to the current example). In the third loop we have a slightly more sophisticated use of symbolic propagation: evaluation of simple symbolic expressions. On the other hand, the subroutine SUB3 accepts an array-element, which is treated as a scalar.

```
PROGRAM TEST
C
COMMON A(100)
DIMENSION B(100),C(100),AA(100),BB(100),CC(100)
READ *,N
IF(N.GT.100.OR.N.LT.1) STOP 'BAD N'
C   PARALLEL LOOP WITH SUBROUTINE CHANGING A COMMON
C   ARRAY.
DO I=1,N
CALL SUB1(I,N)
AA(I)=-A(I)
END DO
C    PARALLEL LOOP WITH SUBROUTINE CHANGING AN ARRAY
C    PARAMETER.
DO J=1,N
CALL SUB2(B,J,N)
BB(J)=-B(J)
END DO
C    PARALLEL LOOP WITH SYMBOLIC ARITHMETIC.
DO K=1,N
KP=K+1
CALL SUB3(C(KP-1),K,N)
CC(K)=-C(K)
END DO
C
```

```
S=0
DO L=1,N
S=S+AA(L)**2+BB(L)**2+CC(L)**2
END DO
PRINT *,S
END
SUBROUTINE SUB1(I,N)
COMMON A(100)
IF(N/2.GT.I) THEN
A(I)=SIN(FLOAT(I))
ELSE
A(I)=COS(FLOAT(I))
END IF
RETURN
END
SUBROUTINE SUB2(B,I,N)
DIMENSION B(N)
IF(N/2.GT.I) THEN
B(I)=SIN(FLOAT(I))
ELSE
B(I)=COS(FLOAT(I))
END IF
RETURN
END
SUBROUTINE SUB3(C,I,N)
IF(N/2.GT.I) THEN
C=SIN(FLOAT(I))
ELSE
C=COS(FLOAT(I))
END IF
RETURN
END
```

Some of the tools parallelized some of the loops, but none could parallelize all of them.

4.2 Utilization of the User's Knowledge

Static analysis tools have an inherent limitation: Not all the crucial expression in a non-trivial program can be evaluated well enough to decide, for every variable at every statement, whether it had been defined or used in that statement. This is the cause of spurious dependences reported by all the analyzers.

All the tools builders recognize the need for some help from the user. Therefore most of the tools allow the user to modify the dependence-list, usually by letting him explicitly delete a dependence. This procedure has several drawbacks:
First — if, due to a modification in the program, it has to be reanalyzed, most of the modifications to the dependence list might be lost, and the programmer will have to redo the analysis of spurious dependences all over again.
Second — deleting a dependence conveys very little information to the analyzer. When the analyzer declared a spurious dependence it is due to a flaw in its analysis, like e.g. an unevaluated IF condition. It is very difficult to deduce the value of the IF condition from the fact that one or more dependences are spurious.
Third — for most Fortran users a dependence is not an intrinsic feature of the program, so that they may spend a lot of time deciding whether a dependence is spurious or not.

We believe that a better way to utilize the user's knowledge is by using assertions and similar constructs which convey information more natural to the programmer, more invariant under changes in the program, and, hopefully, more useful to the tool. These assertions become a permanent part of the program text, thus making it more readable too. A simple preprocessor could translate most of the assertions into Fortran statements, thus checking their validity at run-time and preventing them from becoming "stale". (One of the tools does allow the user to supply more meaningful information, like adding numerical boundaries to do-loops during the interactive stage of the analysis, but this is not a real solution to the problem.)

Some useful assertions would be:

1. assert array_is_defined (`<array-name>, <range1>, <range2>, ...`)

   assert array_is_undefined (`<array-name>, <range1>, <range2>, ...`)

   All the elements of the Array-section `<array>(<range1>,<range2>,...)` have been defined (or declared undefined). (A range is a pair of numeric or symbolic expressions, or, possibly an implied do-loop tripple).

2. assert variable_is_defined (`<variable-name>`)

   assert variable_is_undefined (`<variable-name>`)

3. assert variable_in_range (`<variable-name>, <range>`)

   Variable `<variable>` is in the range `<range>`.

Another useful construct: "Error_Exit", if placed in one of the branches of an `IF` statement will tell the tool that this branch is an error-exit. The tool will then ignore that branch for the purpose of parallelization, and will instruct the compiler to provide for a nice abortion of the program in case the branch will be entered.

The following example illustrates the use of assertions:

```
DO I=1,N
A(COMPLICATED_FUNCTION_OF (I) ) = B(I)
END DO
assert  array_is_defined (A, [0, N-1, 1] )
 .
 .
 .
J = ANOTHER_COMPLICATED_EXPRESSION
```

```
assert  variable_in_range (J, [1, N-2, 2] )
C = C + A(J)
```

Only with the help of both assertions can a tool decide that the array-section $A(1 : N-2)$ had been defined before being used.

A more realistic example of this type is Example III in Appendix IV.

5. CONCLUSION

While there is little doubt that assertions and improved dependence analysis could be useful, it is not clear how much could be gained by such enhancements.

Will a "more powerful" analyzer find significantly less spurious dependences? Will the amount of necessary assertions not be prohibitive ?

In order to get answers to these questions we are designing an experimental analyzer, which, at the price of memory space and CPU time, will perform dependence analysis for discovering loop-carried dependences in parallel loops (step 2 of the parallelization process). The analyzer should recognize the various assertions and other constructs, which will, hopefully, help it do a more accurate analysis.

The purpose of the analyzer is to find out how far should symbolic-propagation be carried out and which types of assertions are useful in order to reduce the number of spurious dependences to a manageable number.

The heart of the analyzer will be a symbolic interpreter, which will propagate symbolic values of integer, logical and character variables from statement to statement, through branches, joint-points and subprograms. The interpreter will also create and propagate status attributes of all the scalar variables and of array segments. Whenever the interpreter will not be able to decide on a parallelization-inhibiting dependence because of lack of information—the relevant information will have to be supplied by the user in the form of an assertion, which will be included in the program text.

By using such a tool to analyze various "real" programs, we believe that one can get some answer to the usefulness of the enhancements.

ACKNOWLEDGEMENTS

This work was supported by the Center for Research in Parallel Computation

We would like to thank Chuck Koebel for very helpful comments on an early version of this paper

APPENDIX I:

Serial dynamic analysis of a parallel loop for loop-carried dependences

Features of a source-to-source compiler for detecting parallelization-inhibiting, loop-carried dependences in a parallel loop.

In addition to loop-carried dependences the compiler also helps detecting "Defined-in-parallel-loop" variables, and "inconsistently used arrays", which contain some elements which are defined in a parallel loop, and other elements which are live in the same loop.

If there is more than one parallel loop, or if a parallel loop is part of a serial loop and is executed more than once, then it may happen that a variable (or an array-element) is "Defined-in-parallel-loop" in one parallel loop, and "Live" in another. This condition is also detected by the compiler.

For the sake of simplicity we assume no nested parallel loops in the program.

The compiler has to add the following code to the program:

For each variable, and each element of each array in the parallel loop:

Global attributes:

1 "Defined/Undefined" bit,

1 "Live/Killed" bit,

1 "Defined/Undefined-in-parallel-loop" bit.

Local attributes:

1 "defined/undefined-in-current-cycle" bit,

1 "live/killed-in-current-cycle" bit,

1 "defined/undefined-in-previous-cycle" bit,

1 "live/killed-in-previous-cycle" bit.

- Initially all variables are "Undefined", "Undefined-in-parallel-loop" and "Killed".
- Whenever there is an assignment to one of these variables outside the parallel loops it becomes "defined".
- On entering the parallel loop all these variables become: "undefined-in-previous-cycle", "killed-in-previous-cycle".
- Beginning each cycle of the parallel loop each of the variables becomes: "undefined-in-current-cycle", "killed-in-current-cycle".
- If one of the variables is DEFINED inside the loop, then:
  - If it is "live-in-previous-cycle" $\Rightarrow$ Read/Write loop-carried dependence.
  - It becomes "defined-in-current-cycle".
- If one of the variables is USED inside the parallel loop, then:
  - If it is "undefined-in-current-cycle" then if it is "defined-in-previous-cycle" $\Rightarrow$ Read/Write loop-carried dependence. It becomes "live-in-current-cycle".
- At the end of each cycle of the parallel loop the attributes of each variable become:
  - "defined/undefined-in-previous-cycle" becomes —

    union of: "defined-in-current-cycle", "defined-in-previous-cycle".
  - "live/killed-in-previous-cycle" becomes —

    union of: "live-in-current-cycle", "live-in-previous-cycle".
- At the end of the parallel loop:
  - If a variable is both "live-in-previous-cycle" and "Defined-in-parallel-loop", or is both "defined-in-previous-cycle" and "Live", then report: Defined in one parallel-loop, Live in another.
  - "live-in-previous-cycle" variables become "Live".

- "defined-in-previous-cycle" variables become "Defined-in-parallel-loop".
- "Defined-in-parallel-loop" variables become "undefined".
- If an array has some "Defined-in-parallel-loop" elements and some "live-in-previous-cycle" elements, $\Rightarrow$ Inconsistently Used Array.
- If an element of an array is "Defined-in-parallel-loop" the whole array becomes "undefined".

- Whenever an "Undefined" variable is being used in the program outside the parallel loops $\Rightarrow$ Either an old bug, or a Parallelization-inhibiting Write/Write loop-carried dependence.

APPENDIX II:

Serial dynamic analysis of the communications in a parallel program

Features of a source-to-source compiler for detecting communication errors in a parallel program.

This compiler uses the results of the dynamic loop-carried-dependence analyzer (Appendix I), therefore it must be used with the same set of data.

Each element of an array which had been partitioned is assigned to one of the virtual processors. Each cycle in each of the parallel loops is assigned to one of the processors.

Each variable and array element of each of the virtual processors gets three attribute bits: "defined/undefined" "defined/undefined-in-communication", "used/unused-in-communication", and one "virtual-processor" integer.

Variables (scalar or whole arrays) are divided , at compile time, into four categories: "partitioned", "local-to-section", "shared" and "spread".

"Partitioned" variables are those which have been explicitly partitioned by the programmer. Arrays in which different elements are defined in different instances of a parallel loop are candidates for partitioning. "Partitioned" variables can be defined and used either in a parallel section or in a communication section.

"Local-to-section" variables are "Defined-in-parallel-loop" variables which are not "partitioned" and are not "Live" in any parallel section. (If the variable is also referenced outside parallel sections then its instances in the parallel sections can be renamed.)

"Shared" variables are those which are not "partitioned", are not "Defined-in-parallel-loop" and if defined by a communication it must be through a COMMUNICATION PRIMITIVE which insures identical results in all processors. These variables are assumed to have identical values in all processors. Only "shared" variables may be used outside parallel and communication sections. In a shared-memory machine "shared" variables can be shared by the processors.

"Spread" variables are all the other variables. They are local to each virtual processor. All the "spread" variables are represented internally as partitioned arrays: A scalar is represented by a one-dimensional array. A one-dimensional array—by a two-dimensional array, etc. The program is compiled accordingly. A "spread" variable can be referenced only in parallel and communication sections.

"local-to-section" variables are distinguished from "spread" for the sake of saving computer memory only.

The compiler must insure that every variable is referenced only according to its category. (This is easily done at compile-time).

Computational Sections

- At the beginning of the program all variables are "undefined".
- When entering a subprogram all the variables which are not SAVEd become "undefined".
- Whenever a variable (or array element) is defined it becomes "defined".
- If an "undefined" variable is being used $\Rightarrow$ Error.
- If any variable (or array element) belonging to one processor is referenced in a parallel loop cycle assigned to another processor $\Rightarrow$ Error.

Communication Section

- In the beginning of each communication section all variables are marked "undefined-in-communication" and "unused-in-communication".

- When a variable is used in the communication section it becomes "used-in-communication". If it is "undefined" ⇒ Error.
- When a variable is defined in the communication section it becomes both "defined-in-communication" and "defined".
- If a variable is both "defined-in-communication" and "used-in-communication" or is defined (in the communication section) while being "defined-in-communication" ⇒ Error.
- If a communication refers to transferring an element to/from an element of a partitioned array in a given processor, then, if that element does not exist in that processor ⇒ Error.

In addition to the parallelization, the tool must also check for internal consistency of the Fortran program: This includes, e.g. relations between formal and actual parameters in subprogram calls, references to array elements, etc.

APPENDIX III:

The set of programs used to test the tools

1. HYD (~ 500 statements, analyzed successfully by two of the tools.)
   - A simple hydrodynamic code which simulates a shock-tube.
   - There is one loop worth parallelizing. It calculates the energy and pressure at each cell, (in a single time-step) by calling a trivial equation-of-state subroutine PCT.
   - The main loop has an internal `GO TO` loop for energy iterations.
   - The program uses one large array to accommodate all the run-time arrays, but there is no real aliasing. This is demonstrated by the following code:

```
COMMON VEC(10000)
READ *,N
IA=1
IB=IA+N
IC=IB+N
.
.
.
CALL SUBR(VEC(IA), VEC(IB), VEC(IC), ..., N)
.
.
.
SUBROUTINE SUBR(A, B, C, ..., N)
DIMENSION A(N), B(N), C(N)
```

2. DYNA (~ 4000 statements)
   - A badly written one-dimensional astrophysical code.
   - One subroutine `MAKEPRF` contains the loops which use almost all of the CPU time, by calling some "heavy" subprograms for calculating equation-of-state, opacities and thermonuclear reaction rates.

- Heavy use of common blocks.

The main difficulties:

- Poor readability.
- Fortran-66 style: Many `GO TO`s.
- Information not localized: Inner subroutines use global arrays unnecessarily.
- Inner subroutines have local initialization and many SAVEd variables.

3. TFSS ($\sim$ 4000 statements)

   Calculation of various thermodynamic quantities for a two-dimensional Thomas-Fermi quasi-molecule in a hot plasma.

- Extensive use of subprograms as parameters to other subprograms.
- Can be parallelized, for a small number of CPUs, in two ways:

  a. Most of its time is spent in calculating several 2-D integrals. These could, in principle, be calculated in parallel.

  b. The program reads many sets of data and calculates them one by one. Computation of different sets of data could be performed in parallel.

4. KWECH: ($\sim$ 400 statements)

   A small program for compression of ASCII data. Reads a block, compresses it, writes the compressed block, and goes back to reading. Can be parallelized for a few CPUs by reading several blocks sequentially, compressing them in parallel, then writing them out sequentially.

APPENDIX IV:

Additional examples of loops which cannot be analyzed by current tools

EXAMPLE Ia.

Same as example one, but DO K loop was modified, to include symbolic expressions both in the loop-header and in the loop-body. The expression in the loop-header contains a variable defined in a DATA statement.

```
DIMENSION A(100),B(10)
DATA KMIN/2/
READ *,N
IF(N.GT.100.OR.N.LT.1) STOP 'BAD N'
C   PARALLEL LOOP
DO I=1,10
C                        DEF LOOP
DO J=1,N
A(J)=J+I
END DO
C -------------------------------
S=0.0
C                        USE LOOP
DO K=KMIN+1,N+1,2
S=S+A(K-1)**I
END DO
C ------------------------------
B(I)=S
END DO
PRINT *,B
END
```

EXAMPLE Ib. Same as example Ia., but loop DO J was replaced by two loops, one defining the odd elements of array A, and one—the even elements. Together the loops span the same segment of array A as the original loop in example Ia.

```
DIMENSION A(100),B(10)
READ *,N
```

```
IF(N.GT.100.OR.N.LT.1) STOP 'BAD N'
C   PARALLEL LOOP
DO I=1,10
C                           DEF LOOPS
DO J=1,N,2
A(J)=J+I
END DO
DO J=2,N,2
A(J)=J-I
END DO
C -----------------------------
S=0.0
C                           USE LOOP
DO K=1,N
S=S+A(K)**I
END DO
C -----------------------------
B(I)=S
END DO
PRINT *,B
END
```

EXAMPLE 3:

Define and use array elements when the index to the array is difficult to evaluate.

The print statement does not depend on the "A(M)=1.0" statement, if we assume that the RANDOM() function returns a REAL value between 0.0 and 1.0.

```
DIMENSION A(100)
READ *,N,M
IF(N.LT.1.OR.N.GT.100) STOP 'BAD N'
IF(M.LT.1.OR.M.GT.100) STOP 'BAD M'
A(M)=1.0
DO I=10, 90
A(I)=0.0
END DO
K=78.0*RANDOM() + 11.0
PRINT *,A(K)
```

REFERENCES

1. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*, published by Prentice Hall, Englewood Cliffs, New Jersey, 1988.

   I. G. Angus, G. C. Fox, J. S. Kim, and D. W. Walker, *Solving Problems on Concurrent Processors, Volume II: Software for Concurrent Processors*, published by Prentice Hall, Englewood Cliffs, New Jersey, 1990.

2. .... A reference to FORTRAN90D ....

3. K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. C. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise and D. Clemmer, 'An Environment for Parallel Structuring of Fortran Programs', in the *International Conference on Parallel Processing*, Volume II, pp. 98-105, 1989.

   (The name E/SP appears in: K. Sridharan, R. Narayanaswamy, C. Denton and B. Eventoff, 'Parallel Structuring of Programs Containing I/O Statements', in the *International Conference on Parallel Processing*, Volume II, pp. 224-228, 1990.)

4. K. Smith, and w. F. Appelbe, 'PAT — An Interactive Fortran Parallelizing Assistant Tool', in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 58-62, August 1988.

5. D. Klappholz, X. Kong, and A. Kallis, 'FATCAT: Fortran Advanced Technology Code Analysis Tool', Technical Report 9102, Stevens Institute of Technology, Hoboken, NJ, January, 1991.

6. V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley and J. Subhlok, 'The Parascope Editor: An Interactive Parallel Programming Tool', in *Proceedings*

of Supercomputing '89, Reno, NV, November 1989. Also, Internal Report Rice COMP TR89-92 May, 1989.

7. C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung and D. Schouten, 'Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors', in *International Journal of High Speed Computing*, 1, 45–72 (1989).
8. P. Havlak and K. Kennedy, *IEEE Transactions on Parallel and Distributed Systems*, **2**, 350 (1991).