PFC:
A Program to Convert Fortran to Parallel Form

by

John R. Allen
Ken Kennedy

Rice University
MaSe Technical Report 82-6

October, 1982
PFC:
A Program to Convert Fortran to Parallel Form

John R. Allen
Ken Kennedy

Department of Mathematical Sciences
Rice University
Houston, Texas 77251

Abstract

The recent success of vector computers like the Cray-1 and array processors such as those manufactured by Floating Point Systems has increased interest in making vector operations available to the Fortran programmer. The Fortran standards committee is currently considering extensions to Fortran which will permit the programmer to explicitly specify vector and array operations. The proposed standard is usually referred to as Fortran 8x.

This paper describes PFC, a system that translates sequential programs written in Fortran to Fortran 8x, replacing loops by array operations wherever possible. Central to the theory underlying PFC is the concept of dependence. In another work [Kenn 80], we developed a test for dependence between statements which distinguished dependences that arise due to the iteration of different loops.

In this work, we show how that test is incorporated into a powerful program for recognizing parallelism. By using a careful implementation strategy, combined with judicious choice of data abstraction mechanisms, we have been able to implement a flexible and sophisticated software system while maintaining reasonable efficiency. In fact, our implementation outperforms one of our previous efforts by a factor of more than ten.

The resulting program is an interesting case study of the application of theoretical techniques to a practical implementation problem.
1. Introduction

With the advent of successful vector computers like the Cray-1 [Cray 76, Russ 78] and the popularity of array processors like the Floating Point Systems AP-120 [FPS 78, Witt 78], there has been increased interest in making vector operations available to the Fortran programmer. One common method is to supply a "vectorizing" Fortran compiler [Cray 80] as depicted in Figure 1. Here standard Fortran is accepted as input and, as a part of the optimization phase of the compiler, an automatic vectorization stage attempts to convert the innermost loops to vector operations. The code generator can then produce vector machine code for these operations.

This scheme has several advantages. Programmers need not learn a new language since the Fortran compiler itself takes on the task of discovering where vector operations may be useful. This also means that there should be no major conversion effort required to bring old code across to the new machine.

In practice, however, there are drawbacks to this system. Because exploiting parallelism in programs is a subtle intellectual activity, compilers aren't sophisticated enough to do a thorough job of recognizing vector operations. As a result, the programmer must assist by re coding loops to trick the compiler into recognizing them as vector operations. Indeed, the Cray Fortran manual [Cray 80] has several pages devoted to such recoding methods. In this environment, the programmer

![Figure 1. A vectorizing Fortran compiler](image-url)
is once again obligated to rewrite his programs for a new machine, not because the compiler will not accept the old program, but because the compiler is unable to convert that program into suitably efficient code. During a number of visits to Los Alamos Scientific Laboratory (which has two Crays), we have observed the widespread sentiment that every Fortran program will need to be rewritten if it is to be acceptably efficient on the Cray-1.

This leads us to the question: if we are forced to rewrite Fortran programs into vector form anyway, why not write them in a language that permits explicit specification of vector operations while still maintaining the flavor of Fortran? Many such languages have been proposed. Vectran [Paul 75, Paul 78] is one of the earliest and most influential of such proposals, although there have been numerous others [Burr 72, DoE 79, Meth 79]. In fact, it seems clear that the next ANSI standard for Fortran, which we shall refer to as Fortran 8x, will contain explicit vector operations like those in Vectran [Paul 80, ANSI 81].

Suppose then, that instead of a vectorizing Fortran compiler we were to provide a Fortran 8x compiler for use with a vector machine. This would allow the programmer to bypass the implicitly sequential semantics of Fortran and explicitly code vector algorithms in a language designed for that purpose. However, a problem remains: what do we do about old code?

One answer is to provide a translator that will take Fortran 66 or 77 as input and produce Fortran 8x as output. This would lead to the system depicted in Figure 2. An advantage of this system is that, since the translation from Fortran to Fortran 8x is done only once or twice, the translator need not be as efficient as a vectorizing stage embedded in a compiler must be. Therefore, the translator might attempt substantially more ambitious program transformations, perhaps using techniques from program verification and artificial intelligence. Presumably, this would enable a more effective translation.

A second advantage is that vector operations missed by the translator can be corrected directly in the Fortran 8x version by the programmer. This is obviously much more desirable than the alternate solution
of recoding input to force the translator to recognize vector operations. This is particularly important since there are some loops which, because of the underlying structure of the problem being solved, can be directly converted to vector form without error, even though the inherently sequential semantics of Fortran makes them difficult or impossible for a translator to convert. Such loops are often easy to recode as explicit vector statements in Fortran 8x.

This paper reports on a project at Rice University to develop an automatic translator from Fortran to Fortran 8x. The Rice project, based initially upon the research of Kuck and others at the University of Illinois [Kuck 77, Kuck 78, KKLN 80, KKLP 81, Mura 71, Leas 76, Bane 76, Towl 76, Wolf 78], is a continuation of work begun at IBM Research in Yorktown Heights, NY. Our first implementation was based on the Illinois Parafrase Compiler [Wolf 78, KKLN 80], but the current version is a completely new program, although many of the translations are the same as those performed by Parafrase. Other projects which have influenced our work are the Texas Instruments ASC compiler [Coha 73, Wede 75], the Cray-1 Fortran compiler [Higb 79], and the Massachusetts Computer Associates Vectorizer [Leve 77, Nysz 78].

The paper is organized into seven sections. Section 2 introduces Fortran 8x and gives examples of its use. Section 3 presents an overview of the translation process along with an extended translation example. In Section 4 the concept of dependence is introduced and a test for loop-carried dependence is described. Section 5 discusses the program
transformations which are preliminary to applying the dependence test, Section 6 presents details on how the test is used for each pair of statements, and Section 7 shows how the resulting dependence information is applied. Section 8 introduces auxiliary transformations that can enhance the opportunities for vectorization. Finally, Section 9 describes the current state of the Rice implementation and discusses future plans.

In a related project, we are implementing a compiler that will accept the output of the translator as input. That project will be reported upon in another paper. Other related papers discuss the theoretical underpinnings of our project [Kenn 80], and a method for handling conditional statements [GibK 81].

2. Fundamentals of Fortran 8x

It is difficult to describe any language whose definition is still evolving, much less write a language translator for it. Nevertheless, we need some language as the basis for our discussion. In this section, we will describe one potential version of the language, which is similar to what the ANSI X3J3 committee is considering. This version is an extension of 1977 ANSI Fortran to include the proposed features for support of array processing and most of the proposed control structures for the language.

2.1. Array Assignment

Vectors and arrays may be treated as aggregates in the assignment statement. Suppose X and Y are two arrays of the same dimension, then

\[ X = Y \]

copies Y into X element by element. In other words, this assignment is equivalent to

\[ X(1) = Y(1) \]
\[ X(2) = Y(2) \]
\[ \vdots \]
\[ X(N) = Y(N) \]
Scalar quantities may be mixed with vector quantities using the convention that a scalar is expanded to a vector of the appropriate dimensions before operations are performed. Thus,

\[ X = X + 5.0 \]

adds the constant 5 to every element of array \( X \).

Array assignments in Fortran 8x are treated as if every input operand is fetched in its entirety before any single output operand is stored. To see the effect of this, consider

\[ X = X/X(2) \]

Even though the value of \( X(2) \) is changed by this statement, the original value of \( X(2) \) is used throughout, so that the result is the same as

\[
\begin{align*}
T & = X(2) \\
X(1) & = X(1)/T \\
X(2) & = X(2)/T \\
. & \\
. & \\
X(N) & = X(N)/T
\end{align*}
\]

This important semantic distinction will have a significant impact on the translation process.

2.2. Array Sections

Individual rows and columns may be assigned using the "\( * \)" notation or the triplet notation. Suppose \( A \) and \( B \) are two dimensional arrays, then

\[ A(*,I) = B(J,*) \]

assigns the \( J \)th row of \( B \) to the \( I \)th column of \( A \). The "\( * \)" notation may be used for single dimension assignment as well, so that

\[ X(*) = Y(*) \]

has the same effect as

\[ X = Y \]

One may define a range of iteration for vector assignment using the triplet notation. For example, suppose you wish to assign the first \( M \) elements of the \( J \)th row of \( B \) to the first \( M \) elements of the \( I \)th column
of A. In Fortran 6x, the following assignment would suffice:

\[ A(1:M,1) = B(J,1:M) \]

This statement would have the effect of the assignments:

\[ A(1,I) = B(J,1) \]
\[ A(2,I) = B(J,2) \]
\[ \vdots \]
\[ A(M,I) = B(J,M) \]

even though \( M \) might contain a value much smaller than the actual upper bound of these arrays.

The term "triplet" seems to imply that the iteration range specifications such as the one above should have three components. Indeed, the third component, when it appears, specifies a "stride" for the index vector in that subscript position. For example, if we had wished to assign the first \( M \) elements of the \( J \)th row of \( B \) to the first \( M \) elements of the \( I \)th row of \( A \) in odd subscript positions, the following assignment might be used.

\[ A(1:M*2-1:2) = B(J,1:M) \]

The triplet notation is also useful in dealing with operations involving shifted sections. For example, the assignment

\[ A(I,1:M) = B(1:M,J) + C(I,3:M+2) \]

has the effect

\[ A(I,1) = B(1,J) + C(I,3) \]
\[ A(I,2) = B(2,J) + C(I,4) \]
\[ \vdots \]
\[ A(I,M) = B(M,J) + C(I,M+2) \]

2.3. **Array Identification**

Useful as it is, the triplet notation provides no way to skip through elements of a rotated array section, like the diagonal. To do this, one must use the IDENTIFY statement, which allows an array name to be mapped onto an existing array. For example,

**IDENTIFY** /1:M/ D(I) = C(I,I+1)
defines the name D(x), for I from 1 to N, as the superdiagonal of C. Thus

\[ D = A(1:N,J) \]

has the effect

\[ C(1,2) = A(1,J) \]
\[ C(2,3) = A(2,J) \]
\[ \vdots \]
\[ C(N,N+1) = A(N,J) \]

It is important to note that D has no storage of its own -- it is merely a pseudonym for a subset of the storage assigned to C.

2.4. Conditional Assignment

Fortran 8x will permit an array assignment to be controlled by a condition array by using the WHERE statement. An example is

\[ \text{WHERE (A.GT.0.0) A = A + B} \]

which specifies that the vector sum of A and B be formed but the store back to A take place only in positions where A was originally greater than zero. The semantics of this statement are such that it behaves as if all of the components of the respective arrays are involved in the computation of the right hand side, but that stores take place only in the positions where the controlling conditions are false.

In the special case of statements like

\[ \text{WHERE (A.NE.0.0) B = B/A} \]

the semantics require that any divide checks arising as a result of evaluating the right hand side be hidden from the user. In other words, any error side effects that might occur as a result of evaluating the right hand side in positions where the controlling vector is false are voided.

2.5. Library Functions

Mathematical library functions, such as SQRT and SIN, are extended on an elementwise basis to vectors and arrays. In addition, new intrinsic functions are provided, such as inner product (DOTPRODUCT) and
transpose (TRANSPOSE). The special function

\[
\text{SEQ}(1,N)
\]

returns an index vector from 1 to N. Reduction functions, much like those in APL, are also provided. For example, SUM applied to a vector returns the sum of all elements in that vector.

2.6. User Defined Subprograms

There are several enhancements to the handling of user-defined subroutines and functions. First, arrays, even identified arrays, may be passed as parameters to subroutines. Second, an array may be returned as the value of a function. And finally, any scalar function may be declared to be elemental; it can then be applied on an elementwise basis to a vector or array to produce a conformable vector or array.

3. The Translation Process

Now we are ready to describe, in an idealized way, the process of translating a Fortran program to Fortran 8x. In so doing, we will illustrate some important aspects of the problem.

Suppose the translator is presented with the following Fortran fragment:

\[
\begin{align*}
\text{DO } 20 \ I = 1,100 \\
\quad \text{KI} = I \\
\quad \text{DO } 10 \ J = 1,300,3 \\
\quad \quad \text{KI} = \text{KI}+2 \\
\quad \quad \text{U(J)} = \text{U(J)}*\text{W(KI)} \\
\quad \quad \text{V(J+3)} = \text{V(J)}+\text{W(KI)} \\
10 & \quad \text{CONTINUE} \\
20 & \quad \text{CONTINUE}
\end{align*}
\]

The eventual aim is to convert statements (3) and (4) to vector assignments, removing them from the innermost loop. That will be possible if there is no semantic difference between executing them in a sequential loop and executing them as vector statements. Consider a somewhat simpler case.

\[
\begin{align*}
\text{DO } 10 \ I = 1,100 \\
\quad \text{X(I)} = \text{X(I)}+\text{Y(I)} \\
10 & \quad \text{CONTINUE}
\end{align*}
\]

If we are to convert this to a vector assignment
we must be sure that no semantic difference arises. Specifically, the main semantic difference in parallel assignment is that the right hand side is assumed loaded before any stores occur on the left. In other words, it uses only old values of its input operands. This might not be equivalent to the sequential loop if that loop computes a value on one iteration and then uses it on a later one, as in the following fragment:

\[
\text{DO 10 } I = 1,100 \\
X(I+1) = X(I) + Y(I) \\
10 \text{ CONTINUE}
\]

This loop cannot be directly converted to the parallel assignment

\[
X(2:101) = X(1:100) + Y(1:100)
\]

because on each iteration after the first it uses a value computed on the previous iteration. The vector assignment would use only old values of \(X\). An iterated statement which depends upon itself in the manner shown is called a recurrence.

In order to distinguish between recurrences and statements that can be directly vectorized, the translator must perform a precise test to determine whether or not a statement depends upon itself - that is, whether or not it uses a value that it has computed on some previous iteration. Details of this dependence test will be provided in the next section; for now it is enough to know that certain program transformations are required to make the test possible.

The first of these, DO-loop normalization, transforms loops so that the loop induction variables iterate from 1 to some upper bound by increments of 1. Sometimes new induction variables must be introduced to accomplish this. Within the loop, every reference to the old loop induction variable is replaced by an expression in the new induction variable. The effect of DO-loop normalization on our example is shown below.
DO 20 I = 1,100
  KI = I
DO 10 j = 1,100
  KI = KI+2
  U(3*j-2) = U(3*j-2)*W(KI)
  V(3*j+1) = V(3*j-2)+W(KI)
10 CONTINUE
  J = 301
20 CONTINUE

(6)

Note that the new variable j (written as a small letter to signify that it has been introduced by the translator) is now the inner loop induction variable and that an assignment (6) has been introduced to define the previous induction variable on exit from the loop.

A major goal of this sequence of normalizing transformations is to convert all subscripts to linear functions of loop induction variables. To accomplish this conversion, references to auxiliary induction variables, such as KI in our example, must be replaced. This transformation, called induction variable substitution, replaces statements that increment auxiliary induction variables with statements that compute them directly using normal loop induction variables and loop constants. The effect in our example is as follows:

DO 20 I = 1,100
  KI = I
DO 10 j = 1,100
  U(3*j-2) = U(3*j-2)*U(KI+2*j)
  V(3*j+1) = V(3*j-2)+W(KI+2*j)
10 CONTINUE
  KI = KI + 200
  J = 301
20 CONTINUE

Here the computation of KI has been removed from the loop and all references to KI have been replaced by references to the initial value of KI plus the sum total of increments that can occur by the relevant iteration, expressed as a function of j. At the end of the loop, an assignment updates the value of KI by the aggregate total of all increments in the loop. Note that since it strives to replace simple additions with multiplications, induction variable substitution is, in a sense, an inverse of the classical optimization technique operator strength reduction [Cock 77, A1CK 81].
The final transformation in preparation for dependence testing is **expression folding**, which substitutes integer expressions and constants forward into subscripts, with simplification where possible. The result in our example:

```plaintext
DO 20 I = 1, 100
   DO 10 J = 1, 100
       U(3*J-2) = U(3*J-2)*W(I+2*J)
       V(3*J+1) = V(3*J-2)+W(I+2*J)
   10    CONTINUE
(5)   KI = I + 200
(6)   J = 301
20    CONTINUE
```

In this example, the first assignment to KI in the outer loop has been removed and references to KI replaced by the right hand side (1) in statements (3), (4), and (5). It should be noted that statement (5) could now be removed by forward substitution, and this is in fact done in the actual translator.

Once the subscripts have been transformed, a standard data flow analysis phase can be applied to build the data flow graph for the whole program. This can be used to propagate constants throughout the program and to recognize **dead statements**, i.e., statements whose output will never be used. PFC's dead statement eliminator is so effective that the accidental omission of an output statement will often lead PFC to produce a null program as output. In the example above, suppose that KI and J are both dead after the code segment shown. Then all assignments of those variables will be deleted, as shown below.

```plaintext
DO 20 I = 1, 100
   DO 10 J = 1, 100
       U(3*J-2) = U(3*J-2)*W(I+2*J)
       V(3*J+1) = V(3*J-2)+W(I+2*J)
   10    CONTINUE
20    CONTINUE
```

The whole point of this complex assortment of transformations is to convert as many subscripts as possible to a canonical form: linear functions of the do loop induction variables. This form makes it possible to apply a powerful and precise test for interstatement dependence. In the example above, we have succeeded in putting all subscripts into the desired form, so we can use the precise test to determine what
dependences exist among the statements in the inner loop.

Once the dependences have been identified, we are ready for parallel code generation. Using dependence information, the translator determines which of the statements remaining does not depend on itself. As it happens, statement (3) does not depend upon itself while statement (4) does (and hence represents a recurrence). Therefore, statement (3) is converted to a vector assignment, while statement (4) is left in a sequential loop by itself.

\[
\begin{align*}
\text{DO } & 20 \text{ I } = 1,100 \\
(3) & \quad U(1:298:3) = U(1:298:3) \times W(I+2:I+200:2) \\
\text{DO } & 10 \text{ j } = 1,100 \\
(4) & \quad V(3\times j+1) = V(3\times j-2)+W(I+2\times j) \\
& 10 \text{ CONTINUE} \\
& 20 \text{ CONTINUE}
\end{align*}
\]

Figure 3 gives an overview of the translation process as implemented in PFC.
Figure 3. Overview of translation.

The scanner-parser phase converts the input program to an abstract syntax tree which is used as the intermediate form throughout the translation. The pretty printer can reconstruct a source program from the abstract syntax tree; it is used throughout the translator. The vector translation phase consists of three main subphases:

1. **Subscript standardization**, which encompasses all the transformations that attempt to put subscripts into canonical form,

2. **Dependence analysis**, which builds the interstatement dependence graph, and

3. **Parallel code generation**, which generates array assignments wherever possible.

Each of these will be discussed in more detail later in the paper. But to understand these phases, you must understand the test for dependence used in FFC. That will be the subject of the next section.
4. **Interstatement Dependence**

In this section we will present the fundamentals of the dependence test used in PPC and illustrate its usefulness. For an extensive treatment of the theory underlying this test, see the earlier work by Kennedy [Kenn 80] and Banerjee [Bane 76].

In general, a statement cannot be rewritten directly into vector form if it depends upon itself. Put another way, if a statement in a loop computes a value that affects one of the same statement's inputs on a later iteration, then that statement cannot be rewritten as an array assignment, because the semantics of Fortran 8x specify that all the inputs of an assignment statement must be extracted before any output is produced. That means that if we rewrite the statement as an array assignment, the array assignment will have a different meaning from the statement it replaces.

Let us consider an example. The loop below has two statements which depend upon each other.

\[
\begin{align*}
\text{DO } & 10 \ j = 1, 100 \\
(1) & \quad X(j) = Y(j) \times A(j) \\
(2) & \quad Y(j+1) = X(j) + B(j) \\
10 & \quad \text{CONTINUE}
\end{align*}
\]

Statement (2) depends on statement (1) because it uses the value computed by statement (1) on the same iteration. Statement (1) depends on (2) because it uses the value computed by (2) on iteration i as input on iteration i+1.

According to our definition, we cannot rewrite these two statements as vector statements because each depends upon itself indirectly through the other. This confirms our intuition, since it is easy to see that

\[
\begin{align*}
(1) & \quad X(1:100) = Y(1:100) \times A(1:100) \\
(2) & \quad Y(2:101) = X(1:100) + B(1:100)
\end{align*}
\]

does not compute the same result as the original loop. The \( Y \) inputs to the first statement are no longer correct.

Kuck and his students have defined three types of dependence that can hold between statements [Kuck 78, Towl 76]:

- 15 -
Definition: If control flow within a program can reach statement \( S_2 \) after passing through \( S_1 \), then \( S_2 \) depends on \( S_1 \) if

(1) \( S_2 \) uses the output of \( S_1 \) (true dependence, written \( S_1 \) \( \rightarrow \) \( S_2 \)).

\[
S_1: \quad X = 6 \\
S_2: \quad = X
\]

(2) \( S_2 \) might wrongly use the output of \( S_1 \) if they were reversed in sequence (antidependence, written \( S_1 \) \( \rightarrow \) \( S_2 \))

\[
S_1: \quad = X \\
S_2: \quad X = \theta^{-1}
\]

(3) \( S_2 \) recomputes the output of \( S_1 \), and if they were reversed, subsequently executed statements might use the wrong value of that output (output dependence, written \( S_1 \) \( \rightarrow \) \( S_2 \)).

\[
S_1: \quad X = \theta^0 \\
S_2: \quad X = \theta^0
\]

All three dependences must be considered when we are trying to detect recurrences. Antidependences prevent us from introducing new dependences by vectorizing, while output dependences preclude us from vectorizing through a scalar bottleneck:

\[
\text{DO 10 I = 1, 100} \\
\text{T = A(I) + 10} \\
\text{B(I) = T - C(I)} \\
\text{10 CONTINUE}
\]

If we vectorized these statements, we would get:

\[
\text{T = A(1:100) + 10} \\
\text{B(1:100) = T - C(1:100)}
\]

which is clearly incorrect because the last value of \( T \) will be used in all computations of \( B \). The output dependence of the first assignment on itself (due to \( T \)) prevents this vectorization. FFC builds dependence graphs with all three kinds of dependences.

In view of these observations we may correctly conclude that, once the dependence graph has been constructed, the task of identifying which
statements can be rewritten into array assignments is equivalent to the
problem of finding strongly connected regions in the dependence graph.
However, we must be careful in doing this because partial vectorization
is possible in many cases. To see this consider the more complicated
example in Figure 4.

```
DO 30 I = 1, 100
   DO 20 J = 1, 100
      DO 5 K = 1, 100
         X(I,J,K) = A(I,J,K) + D
      CONTINUE
   5 CONTINUE
   DO 10 L = 1, 50
      A(I+1,J,L) = X(I,J,L)*C
   10 CONTINUE
20 CONTINUE
30 CONTINUE
```

Figure 4. Layered dependence example.

In this example, statements (1) and (2) depend upon one another, so one
might assume that vectorization is impossible. But that is not strictly
ture. While the two statements cannot be vectorized in all three dimen-
sions they can be vectorized in two dimensions. The problem with our
view of dependence is that it is too monolithic -- a recurrence at any
level inhibits vectorization at all levels.

For this reason, we introduce a layered view of dependence, in
which a dependence is directly attributed to the loop whose variation
gives rise to it. We intend to examine this idea as it applies to our
example, but first we need to define our concept of layered dependence.

**Definition:** Suppose two statements

\[ S_1: X(f(i_k)) = R \]
\[ S_2: L = F(X(g(i_k))) \]

(where \( R \) and \( L \) denote arbitrary right and left hand sides resep-
tively and \( F \) denotes an arbitrary expression in its argument) are
contained in the same loop at nesting level \( k \), and suppose that \( i_k \)
is the loop induction variable at that level. (By convention, nesting
levels are numbered from the outside in, so the outermost loop will
be at level 1. Suppose also that $S_1$ precedes $S_2$ in the loop. Then $S_2$ depends on $S_1$ at level $k$, written

$$S_1 \leq_k S_2$$

if there exist integers $j_1, j_2$ such that

$$1 \leq j_1 \leq j_2 \leq N_k$$

where $N_k$ is the upper bound of the level-$k$ loop, and such that

$$f(j_1) = g(j_2)$$

In other words, if $S_1$ computes a value on one iteration of the loop at level $k$ that is used as input by $S_2$ on the same or a later iteration of that loop, then $S_2$ depends on $S_1$ at level $k$.

In Figure 4, the loop on $I$ is at level 1, the loop on $J$ is at level 2 and the loops on $K$ and $L$ are at level 3. Now statement (2) cannot depend on (1), or vice versa, at level 3 because (1) and (2) are not contained in the same level 3 loop. This confirms the intuition that no amount of variation in the level three loops can give rise to a dependence between these two statements. Thus, the statements can be vectorized in the level 3 dimension.

At level 2, with $K$ and $L$ chosen to be equal, statement (2) depends on (1) through the reference to $X$. But (1) can never depend on (2) at this level because the values of the outer loop subscript in the two references to $A$, can never be equal without varying the outer loop. In other words, variation in $J$ alone cannot make (1) depend on (2). Therefore the dependence of (1) on (2) is at level 1. In other words,

$$S_1 \leq_2 S_2$$

and

$$S_2 \leq_1 S_1.$$

Now since a chain is only as strong as its weakest link, the recurrence involving (1) and (2) is broken if we break the level 1 dependence. Therefore, if we run the level 1 loop sequentially, we can generate vector code for the two innermost dimensions. The final code for Figure 4 will be as follows.
DO 30 I = 1, 100
    X(I,1:100,1:100) = A(I,1:100,1:100) + D
    A(I+1,1:100,1:50) = X(I,1:100,1:50)*C
30 CONTINUE

All this suggests the following recursive scheme for parallel code generation.

(1) Find all strongly-connected regions in the dependence graph.

(2) For any statement S not in a strongly-connected region, generate parallel code in as many dimensions as possible.

(3) For every strongly-connected region R, generate a sequential DO loop to surround it and try to generate parallel code for the statements in R using a dependence graph in which the top-level dependences have been cast out. If parallel code can be generated for such a statement, it will be parallel in one fewer dimension.

The parallel code generation routine \texttt{coderen}, more precisely specified in Figure 5, is almost exactly the one implemented in PFC. It can be easily shown that this procedure will generate statements which are parallel in the maximum number of dimensions consistent with the given dependence graph.
procedure codegen (R, k, D);

/* R is the region for which we must generate code */
/* k is the minimum nesting level of possible parallel loops */
/* D is the dependence graph among statements in R */

find the set \{S_1, S_2, \ldots, S_m\} of maximal strongly-connected regions in the dependence graph D restricted to R (use Tarjan's algorithm);

construct \(P_m\) from \(R\) by reducing each \(S_i\) to a single node and compute \(D_m\), the dependence graph naturally induced on \(P_m\) by \(D\);

let \(\{\pi_1, \pi_2, \ldots, \pi_m\}\) be the \(m\) nodes of \(P_m\) numbered in an order consistent with \(D_m\) (use topological sort to do the numbering);

for \(i = 1\) to \(m\) do

if \(\pi_i\) is strongly-connected

then
generate a level-\(k\) DO statement;

let \(D_i\) be the dependence graph consisting of all dependence edges in \(D\) which are at level \(k+1\) or greater and which are internal to \(\pi_i\);

codegen (\(\pi_i, k+1, D_i\));

generate the level-\(k\) CONTINUE statement

else

generate a parallel statement for \(\pi_i\) in \(\rho(\pi_i)-k+1\) dimensions, where \(\rho(\pi_i)\) is the number of loops containing \(\pi_i\);

fi

od

end

Figure 5. Parallel code generation routine.

In order to use this method for dependence analysis, we must have a good test for dependence at level \(k\). Such a test is given by the theorem below.
**Theorem:** Assume the statements

\[ S_1: \quad X(f(i_1, i_2, \ldots, i_{n_1})) = k \]
\[ S_2: \quad L = F(X(g(i_1, i_2, \ldots, i_{n_2})) \]

occur together in at least \( k \) loops. (Here \( S_1 \) is contained in \( n_1 \) loops and \( S_2 \) is contained in \( n_2 \) loops.) If \( S_1 \) precedes \( S_2 \) in the level-\( k \) loop and \( f \) and \( g \) are linear functions of the loop induction variables, i.e.

\[ f(i_1, i_2, \ldots, i_{n_1}) = a_0 + \sum_{j=1}^{n_1} a_{ij} i_j \] and

\[ g(i_1, i_2, \ldots, i_{n_2}) = b_0 + \sum_{j=1}^{n_2} b_{ij} i_j \]

then \( S_1 \) \( \preceq_k \) \( S_2 \) only if

a) **gcd test:**

\[ \gcd(a_1-b_1, a_2-b_2, \ldots, a_{k-1}-b_{k-1}, a_k, \ldots, a_{n_1}+b_1, \ldots, b_{n_2}) \mid b_0-a_0 \]

b) **Banerjee inequality:**

\[ \begin{align*}
&\sum_{i=1}^{k-1} (a_i-b_i)(N_i-1) - \sum_{i=k+1}^{n_1} a_i(N_i-1) - \sum_{i=k+1}^{n_2} b_i(N_i-1) \\
&\leq \sum_{i=0}^{n_1} a_i i - \sum_{i=0}^{n_2} b_i i \\
&\leq \sum_{i=1}^{k-1} (a_i-b_i)(N_i-1) + \sum_{i=k+1}^{n_1} a_i(N_i-1) + \sum_{i=k+1}^{n_2} b_i(N_i-1)
\end{align*} \]

where \( N_k \) denotes the upper bound for the level-\( k \) loop.

If \( S_2 \) precedes \( S_1 \) then the inequality is slightly different [Kenn 80].

This test provides a precise, though not exact test for dependence between two statements. It is complicated, but its computation can be speeded up by a number of tricks which are described in Section 6.
Efficiently applying this test has been the goal of much of the programming effort in PFC. The effort to get all subscripts into linear form (subscript standardization) is described in the next section. Section 6 describes how this test is incorporated into a pass which analyzes each pair of statements for dependence. Finally, the use of the layered dependence edges to generate code is described in more detail in Section 7.
5. Subscript Standardization

The STANDARD phase of FPC is responsible for translating a program into a normal form that makes dependence testing easier to perform. The primary aim of this phase is to transform each subscript in the program to a linear function of primary loop induction variables, where primary induction variables run from 1 to some upper bound in increments of 1. STANDARD consists of two main transformation modules: STDDNRM which normalizes DO loops and STIVSUE which replaces references to auxiliary induction variables by expressions in the loop induction variable.

5.1. DO Loop Normalization

DO loop normalization is a relatively straightforward process. STDDNRM simply walks the abstract syntax tree (henceforth referred to as the AST) that represents the input program, transforming loops as it encounters them. Whenever it finds a DO statement, it saves the control variable of that loop in an array called the LCD, which is indexed by nesting level. Since STDDNRM intends to replace the DO statement with a standardized version, it creates a variable to control the new loop. The name of this variable, which is formed by concatenating '6' with the nesting level of the statements in the loop, identifies it as a loop induction variable and makes it easy to determine the level of iteration it controls.

The standardized DO statement is now simple to generate. Both the lower bound and the increment are known to be 1; the new upper bound is simply:

$$(\text{OLD\_UPPER\_BOUND} - \text{OLD\_LOWER\_BOUND} + \text{OLD\_STEP}) / \text{OLD\_STEP}.$$ 

Throughout this section, we will refer to the following code fragment to illustrate the transformations we are describing.

(1) \hspace{1cm} \text{DO 100 I = 1, N, K}
(2) \hspace{1cm} \text{DO 50 J = I+1, N}
(3) \hspace{1cm} A(I,J-1) = B(J,I-1) + 10
(4) \hspace{1cm} 50 \hspace{1cm} \text{CONTINUE}
(5) \hspace{1cm} 100 \hspace{1cm} \text{CONTINUE}

STDDNRM will translate DO statement (1) in this example to

\text{DO 100 } &1 = 1, (N-1)/K + 1
As STDDNRN walks the body of the loop, it replaces all references to the old loop control variable with an expression involving the new loop control variables. This expression is stored in the LCD with the old control variable, and is equal to:

\[ \text{OLD_LOWER_BOUND} + \text{OLD_STEP} \times (\text{NEW_LOOP_VARIABLE} - 1) \]

As STDDNRN passes a variable in a loop it checks the variable against all entries in the LCD; if it finds the variable, it replaces that reference with the corresponding expression from the LCD. Note that some care must be taken to replace all the induction variables in a DO statement before generating any new expressions from that DO.

Whenever STDDNRN substitutes inside a subscript, it marks the subscript as one to be later simplified by STDSIMP, a powerful integer expression simplifier which leaves expressions in a canonical form suitable for easy recognition of the terms involved in the dependence tests.

After the substitution and simplification, the example loop has changed to

1. \( \text{DO 100 } \&1 = 1, (N-1)/K + 1 \)
2. \( \text{DO 50 } \&2 = 1, N-\&1\times K + K \)
3. \( A(\&1\times K + K + 1, \&1\times K + \&2 + K) = 5(\&1\times K + \&2 + K + 1, \&1\times K - K) + 10 \)
4. \( 50 \text{ CONTINUE} \)
5. \( 100 \text{ CONTINUE} \)

As STDDNRN leaves a loop, it performs two actions. First, it creates an assignment statement to assign the old loop variable the value it would have retained upon termination of the loop. This assignment is placed after the body of the loop. Second, STDDNRN erases the entry in the LCD for that loop, so that variables will not be accidentally replaced in the wrong contexts. The final code is shown below.

1. \( \text{DO 100 } \&1 = 1, (N-1)/K + 1 \)
2. \( \text{DO 50 } \&2 = 1, N-\&1\times K - K + 1 \)
3. \( A(\&1\times K - K + 1, \&1\times K + \&2 + K) = 5(\&1\times K + \&2 + K + 1, \&1\times K - K) + 10 \)
4. \( 50 \text{ CONTINUE} \)
4.1 \( J = N + 1 \)
5. \( 100 \text{ CONTINUE} \)
6. \( I = N + 1 \)

Note that the transformations performed by STDDNRN do not produce semantically equivalent programs in all cases. In particular, branches
out of loops might cause the old loop control variable to be undefined at the branch target since it is essentially undefined throughout the execution of the standardized loop. While this particular problem can be handled (by creating the assignment to the old variable as the first statement in the loop rather than outside the loop), it is indicative of a more fundamental problem in vectorization — namely, the effects of control flow on data dependence.

The process of vectorization is based upon knowing precisely the data dependences of a group of statements. This information is extremely difficult to determine in the presence of conditional branches, since the control flow can never be known exactly without executing the program (at least symbolically). For example, consider the problems of generating parallel code for a statement in a loop that also contains a conditional branch out of the loop. The parallel statement generated must somehow account for the branch in its upper bound, since the conditional branch could be taken on any iteration of the loop. Obviously, this is a difficult construction for which to generate parallel code. Because of this, PFC restricts its vectorizing efforts to what we call extended basic blocks.

An extended basic block is a section of code that contains no branches other than DO loops. By limiting its efforts to extended blocks, we have been able to avoid the problems that branches cause for dependence analysis while dramatically improving the running time of PFC.

A major disadvantage of restricting our attention to extended blocks is that we may miss the opportunity to convert some statement which is not affected by the branches within that block. This effect is softened somewhat by the use of if conversion, a transformation that aggressively attempts to eliminate as many branches as possible from the program. This transformation is discussed in Section 8.

5.2. Induction Variable Substitution

Initially, STIVSUB was designed only to substitute for auxiliary induction variables of the forms

- 25 -
Var = Var + constant

and

Var = Var - constant.

However, the method we devised for handling substitution was very easily generalized to perform limited constant propagation, expression folding, code motion, and dead code elimination. This has contributed tremendously to the amount of vectorization achievable in PFC, because one of the main impediments to such transformations in our system is that it currently does not test for dependence symbolically. The constant propagation accomplished by STIVSUB replaces the majority of symbolic subscripts in most user programs. Additionally, by eliminating some dead code, STIVSUB decreases the running time of dependence analysis, which dominates the cost of vectorization.

Induction variable substitution is accomplished in two passes over the code. The first pass creates two lists: one of all the variables used as inputs in the block of code and one of all the variables produced as output. Information stored in the input list includes

(1) the variable's symbol table location,

(2) a pointer to the variable's predecessor in the tree (so that it can be replaced),

(3) a pointer to the expression that must be simplified if this variable is replaced,

(4) the relative statement number of the statement in which the reference occurs, and

(5) the nesting level of that statement.

The output list holds

(1) the variable's symbol table location,

(2) a pointer to the predecessor of the statement in which the variable is defined (so that the definition can be deleted if the assignment turns out to be dead),

(3) the relative statement number of the statement, and
(4) the nesting level of the statement.

In both lists, the variables are stored in the order that they are encountered when walking the AST.

If STIVSUB encounters a nested DO statement while building up its lists, it invokes itself recursively on that loop before continuing. Thus, complete substitution for any nested loops will be accomplished before substitution is attempted for the present loop. Consider the following code as an example:

```
KI = 100
1 DO 20 &1 = 1,10
 2 A (KI) = A(KI) - 1
 3 DO 10 &2 =1,20
 4 KI = KI + 1
 5 B (KI) = B (KI) + 1
10 CONTINUE
20 CONTINUE
```

If STIVSUB were called on this section of code, it would initially build up variable lists for statements 1 and 2 (the assignment to KI is outside the extended block). At that point, it would invoke itself recursively on statement 3. When the second invocation returned, the program would be as follows:

```
KI = 100
1 DO 20 &1 = 1,10
 2 A (KI) = A (KI) - 1
 3 DO 10 &2 = 1,20
 5 B (KI + &2) = B (KI + &2) + 1
10 CONTINUE
15 KI = KI + 20
20 CONTINUE
```

At this point STIVSUB would continue building lists for the rest of the program. Note that the initial invocation of STIVSUB would not have to build up the input output lists for statements 3, 5, and 10, because the recursive invocation would have returned the lists for the code it modified. The initial invocation can simply merge the returned lists into its own, thereby avoiding the expense of walking code a second time.
After induction variable substitution is completed, the code will be as follows:

```
1    KI = 100
2    DO 20 I1 = 1, 10
3    A (KI + 20 * I1 - 20) = A (KI + 20 * I1 - 20) - 1
4    DO 10 I2 = 1, 20
5    B (KI + 20 * I1 - 20 + I2) = B (KI + 20 * I1 - 20 + I2) + 1
10   CONTINUE
20   CONTINUE
20   KI = KI + 200
```

Constant propagation using use-definition chains could then substitute the value 100 for all uses of KI, making all the array assignments above very easy to vectorize.

Once the data lists are built for a section of code, the substitution process is a straightforward two-phase procedure that is applied to every member of the output list defined at the outermost nesting level under consideration. The first phase applies the following tests to determine what type of substitution may be performed for the variable:

(1) Check that the statement defining the variable is an assignment and that the variable is not an array or a loop control variable. If any of these conditions are true, we cannot safely perform any substitution.

(2) See if any of the input variables to the assignment statement appear in the loop output list. If they don't, then the assignment is of the form

\[
\text{Var} = \text{loop invariant expression}
\]

and the variable is constant throughout the loop. Therefore the right hand side of the assignment can be forward substituted for occurrences of the variable in the loop. The actual substitution process is described later.

(3) If the only input variables to the statement that appear in the loop output list are loop control variables,

\[
\text{Var} = f(I, \text{loop invariant expressions})
\]

then expression folding can be accomplished in a manner analogous
to loop invariant expressions.

(4) Finally, if the statement defines an auxiliary induction variable,

\[ \text{Var} = \text{Var} \pm \text{loop invariant expression} \]

induction variable substitution can be performed. If there are no other definitions, then an expression involving the loop induction variable can be substituted for all uses of the auxiliary variable throughout the loop. The example earlier in this section demonstrates this type of substitution.

The second phase is the actual substitution process. It involves determining both the extent of code over which substitution may be performed and the actual expression that will be substituted. Since the substitution method for induction variables differs from the method for substituting loop invariant expressions and functions of the loop control variable, we will describe these methods separately below.

Here are the guidelines for induction variable substitution:

(1) If there are no other outputs of the induction candidate within the loop, then all uses of the variable within the loop can be replaced and the assignment statement can be moved outside the loop. This is standard induction variable substitution, and is illustrated in the example earlier in this section.

(2) If the induction candidate is killed later in the loop by a statement at the same nesting level as the candidate definition, then the right hand side of the assignment may be substituted for all uses of the induction variable between the assignment and the kill. Furthermore, the assignment can be safely deleted. For example, in the following code

\[
\begin{align*}
\text{DO 10 } & \text{ I = 1, 10} \\
& 1 \quad \text{K1 = K1 + 1} \\
& A (K1) = A (K1) + 1 \\
& K1 = K1 + 2 \\
& 10 \quad \text{CONTINUE}
\end{align*}
\]

the right hand side of statement 1 may be propagated and the statement deleted to give the following:
DO 10 KI = 1, 10
   A (KI + 1) = A (KI + 1) + 1
   KI = KI + 3
10 CONTINUE

Note that these transformations allow full induction variable substitution to be performed for the remaining assignment to KI. Also note that if the kill is at a deeper nesting level, as in the following, substitution is impossible.

DO 10 &1 = 1, 10
   KI = KI + 1
   A (KI) = A (KI) + 1
   DO 5 &2 = 1, 5
   KI = KI + A(&2)
5 CONTINUE
10 CONTINUE

Since the first assignment cannot be deleted, there is no safe way to substitute for the variable without some form of code motion. We feel that the difficulties involved in performing this type of substitution are not worth the time and effort involved in making them.

(3) If neither of the first two conditions hold, we cannot substitute and we leave the code alone.

Here are the guidelines for substitution of loop invariant expressions (and expressions in loop control variables):

(1) If the next output of the candidate is at the same nesting level as the assignment (or doesn't exist at all), then we may substitute for all uses of the candidate between the assignment and the next output. For example, we can transform

DO 10 &1 = 1, 10
   KI = 3 * &1
   A (KI) = A (KI) + 1
   KI = KI + 1
10 CONTINUE

into the following:
DO 10 61 = 1, 10
A (3 * 61) = A (3 * 61) + 1
KI = 3 * 61 + 1
10 CONTINUE

The disposition of the assignment statement that defines the candidate depends upon several things. If the next output of the variable is definitely kills the first assignment, then the assignment may be deleted (as above). If there is no other output of the variable farther down in the loop, and if the propagation eliminates all inputs of the variable in the loop, the assignment may be modified and moved out of the loop. This transformation is very important, since STDDNRM creates many assignments like the following:

DO 30 61 = 1, 10
   DO 20 62 = 1, 10
      DO 10 63 = 1, 10
         Statements
10 CONTINUE
   K = 11
20 CONTINUE
   J = 11
30 CONTINUE
   I = 11

After application of the previous transformation, this code becomes

DO 30 61 = 1, 10
   DO 20 62 = 1, 10
      DO 10 62 = 1, 10
         Statements
10 CONTINUE
20 CONTINUE
30 CONTINUE
   K = 11
   J = 11
   I = 11

which is obviously a preferred form. If neither of these conditions are met, the assignment must remain in place.
(2) If there is another output of the variable at a deeper level than the original assignment, the right hand side of the assignment may be substituted forward in a limited manner. Uses of the variable after the assignment may be replaced until one of two conditions occurs: either the next definition is reached, or a use at a deeper nesting level than the original assignment is reached. In either case, the assignment cannot be deleted or moved. The following code demonstrates this type of substitution.

```plaintext
DO 20 61 = 1, 10
   KI = 3 * 61
   A (KI) = A (KI) + 1
   DO 10 62 = 1, 10
      B (KI) = B (KI) + 1
      KI = KI + A (62)
10    CONTINUE
20    CONTINUE
```

The first definition of KI may be propagated to the statement immediately following it without danger. However, any further propagation of KI would produce semantically incorrect code. The final code would be as follows:

```plaintext
DO 20 61 = 1, 10
   KI = 3 * 61
   A (3 * 61) = A (3 * 61) + 1
   DO 10 62 = 1, 10
      B (KI) = B (KI) + 1
      KI = KI + A (62)
10    CONTINUE
20    CONTINUE
```

The procedure defined above does not substitute as fully as is possible. In particular, expressions could be substituted for all uses of a variable up to the outermost loop containing the assignment that kills that variable. The above procedure only substitutes to the first nested loop, even though it could safely substitute in all loops prior to one that contains the new definition. Because these types of substitutions arise infrequently and would be relatively expensive to implement with complete safety, we opted for the less complete substitution.
Whenever substitution or deletion occurs, the lists are modified to reflect the changes. This allows some substitution to occur farther down in the loop that would not be possible otherwise. Additionally, nodes that hold substituted expressions are marked so that STDSIMP can normalize them later. After this normalization, the code is standardized and ready for dependence testing.
6. Dependence testing

6.1. Overview

The objective of dependence analysis is to test every relevant pair of statements in a program at all possible levels of dependence. Thus, dependence testing is inherently expensive for at least three reasons:

1. each of the roughly $N^2$ pairs of statements (for a program with $N$ statements) must be examined for dependence,

2. each pair must be tested once for each of the loops that surround both statements, and

3. both of the tests for dependence are relatively expensive to perform.

Because we expected the cost of dependence testing to be high, we focused a large amount of effort on optimizing the time spent testing for dependence. The effort appears to have been well placed; PFC spends more time scanning and parsing a typical Fortran program than it spends performing dependence analysis.

The general algorithm used to implement dependence analysis is given in Figure 6. The following issues are very important in any efficient implementation of the algorithm:

1. A method must be devised to compare each pair of statements in the program for dependence. Because of the expense of applying the layered dependence tests to a new pair of statements, PFC must avoid redundant or unnecessary comparisons. That is, it should consider a pair of statements only once, making all tests for dependences of different kinds at different layers at the same time. Additionally, it should also allow the upper bounds of all loops surrounding either of the statements to be easily found, since this knowledge is required for Banerjee's test.

2. A method must be devised to find all the variables used and defined by a statement. Additionally, this method should provide a way for any given subscript to quickly find the terms involved in the two dependence tests.
for each pair of statements that can be dependent do
    while there are references to the array $A$ in both statements do
        for each level $L$ at which the pair might be dependent do
            /* the variable INDEPENDENT will remain false until the two */
            /* statements can be shown to be independent for $A$ at */
            /* this level */
            INDEPENDENT + false;
            while not INDEPENDENT and
                there is an untested subscript position $i$ of $A$ do
                if the expressions in position $i$ of $A$ in both statements
                    are linear (if they are not, we cannot show independence in
                    this subscript position, so we go on to the next)
                    then
                        if the gcd test shows independence or
                            Banerjee's inequality does not hold
                        then
                            INDEPENDENT + true
                        fi
                    fi
                od
            /* Here if INDEPENDENT is true, we have shown the statements */
            /* to be independent for $A$ at level $L$, otherwise we must */
            /* assume that a dependence exists for $A$ at level $L */
            if not INDEPENDENT then record the level $L$ dependence fi
        od
    od
od

Figure 6. Algorithm for dependence testing.

(3) Finally both Banerjee's test and the gcd test must be carefully
    implemented to be as efficient as possible.

The rest of this section is partitioned to deal with each of the above
issues. Section 2 describes a method for comparing each pair of state-
ments in the program at the correct levels with no redundancy. The
order of comparison generated by this method makes it easy to keep track
of the upper bounds of each loop surrounding the statements. Section 3
describes the method used to store the input and output variables for
each statement. Stored with the variables are the terms that are needed
for Banerjee's test and the gcd test. Section 4 describes the actual implementation of Banerjee's test and the gcd test. Section 5 briefly describes how the parts fit together and mentions some key design decisions that influenced the entire dependence testing phase.

6.2. Statement Comparison

Comparing every pair of statements at all levels at which a dependence could exist is not so easy a problem as it initially appears. The brute force methods for generating comparisons are inelegant and inefficient. Additionally, maintaining the proper set of loop upper bounds can be very messy. We decided to avoid these difficulties by creating a data structure to encode the nesting of the statements in a program. This recursive data structure, DDSTMTLST, is a threaded list structure that allows quick recognition of all loops surrounding any two statements. Although DDSTMTLST is complicated, it is easy to build and it is the basis for an elegant, systematic method of comparing statements.

In DDSTMTLST every statement in a given loop is represented explicitly in a list. This list has a sublist for every nested DO statement that the loop contains. These sublists are identical in structure to the base list; that is, they are lists of statements with sublists for each nested DO statement inside them. To aid the statement comparison process, the sublists of a given list are threaded by creating a new list (a list of sublists) which holds the locations of all sublists of that list. For easy reference, a pointer to the list of sublists for a specific loop is placed in the first node of that statement list for the loop.

An example should clarify the statement list structure. Consider the following program template.
DO
  Statement
  DO
  Statement
  ENDDO
  DO
  Statement
  ENDDO
  Statement
  ENDDO

The list structure created for that program is given in Figure 7. Note that the highest level list (DDSTMTLST) has three nodes: one * node which holds a header to the list of sublists (S1) for DDSTMTLST; the node for statement 1 (the only statement in the block at nesting level 0); and the * node for statement 1 which holds a header to the list (SUBL) of all statements nested inside the outer DO statement. SUBL has two * nodes, one for each of the two DO loops nested within its control. In basic structure, however, it is identical to DDSTMTLST. Likewise, so are SUBL2 and SUBL3.

Only DDSTMTLST and SUBL contain a list of sublists, since they are the only lists with nested DO loops. These lists of sublists (S1 and S2) enable quick location of all the nested loops within a DO loop. For example, DDSTMTLST contains one DO loop. All the statements within that DO may be accessed by following all the lists pending from nodes of S1. Since S1 can be found by looking in the first node of DDSTMTLST (the * node), it takes very little time to find any particular list nested within DDSTMTLST. Similarly, the two lists of nested statements for SUBL can be accessed by following list S2.

Because of its recursive structure, DDSTMTLST is very easy to construct using a stack-oriented procedure. DDBLDSL is the PFC procedure that builds DDSTMTLST; it walks the AST while maintaining two lists: a list of all statements encountered at the current level and a list of all sublists (i.e. a list of pointers to all the lists of nested lists) encountered at this level. As DDBLDSL walks a statement, it adds that statement's number to the present statement list. Whenever a DO is encountered, DDBLDSL pushes the present pair of lists onto a stack and continues walking, with a new, empty pair of lists. When it finds the
Figure 7. DST:TLST for sample program.
end of a DO loop, DDBLDSL carefully pops off the last pair of lists and stores pointers to the lists just completed in the newly popped lists. While this procedure requires attention to ensure that a pointer to a list is not accidentally lost, its implementation is straightforward. The algorithm can be recursively applied at each level of nesting.

A recursive data structure such as DDSTMTLST suggests that a recursive procedure might be used for comparing statements. DDTEST, which is outlined in Figure 8, is such a procedure. Initially it is called at level 0 with DDSTMTLST as a parameter. DDTEST then moves through the statement list, calling a routine COMPARE which actually compares two statements at all levels between 0 and LEVEL. Note that DDTEST is both iterative and recursive; it iterates to move through lists and calls itself recursively to move to deeper nesting levels. The order in which the comparisons would be made in the sample program is given in Figure 9.

Although it is not obvious, DDTEST is optimal in the number of statement comparisons performed. Additionally, the list structure clearly defines entries into nested regions; thus, the bounds of the current loops surrounding a pair of statements can be easily maintained with a stack. DO statements themselves are also easy to locate with DDSTMTLST, since the first statement in every list (other than the outer list) immediately follows a DO.

DDBLDSL performs a minor transformation on the program as it builds the statement list. All DO statements are converted to assignments of the form:

\[
generated\_variable = \text{upper bound expression for the DO.}
\]

The generated variable is similar to those created by STDDVNM; i.e. it clearly reflects the fact that it is artificially generated and it indicates the level of the loop that it controls. The program structure is not lost by this transformation, because the nesting level of all statements has been preserved before the call to DDBLDSL. Additionally, DDSTMTLST can be used to recover the nesting level of any particular statement. There are two distinct advantages to transforming the
procedure DETEST (LIST, LEVEL);
  
/* LIST is the list of statements to be compared at all nesting */
/* levels up to and including LEVEL */
/* First compare all statements in a nested DO loop at a deeper */
/* level. */

for each list SUB on the list of sublists of LIST do
  DETEST (SUB, LEVEL + 1).
od;

/* Next compare every statement on this list with every statement */
/* contained in this do loop at level LEVEL. */

for each statement S1 in LIST do
  for each statement S2 later than or equal to S1 in LIST do
    /* COMPARE does the actual work involved in comparing */
    /* two statements at levels 0, 1, ..., LEVEL */
    COMPARE (S1, S2, LEVEL)
  od
od

/* Next compare all statements in nested loops with every */
/* statement at the top level. */

for each list L1 on the list of sublists for LIST do
  for each statement S1 in L1 do
    for each statement S2 in LIST do
      COMPARE (S1, S2, LEVEL)
    od
  od
od
od

/* Finally compare all statements in different nested loops */
/* at LEVEL */

for each list L1 on the list of sublists for LIST do
  for each statement S1 in L1 do
    for each list L2 beyond L1 on the list of sublists
      for LIST do
      for each statement S2 in L2 do
        COMPARE (S1, S2, LEVEL)
    od
  od
od
od
end

Figure 8. Algorithm for Statement Comparison.
DDTEST (DDSTIMLIST, 0)
calls DDTEST (SUB1, 1)
calls DDTEST (SUB2, 2)
  there are no sublists to invoke DDTEST on.
  compares statements 4 and 4 at level 2.
  returns.
calls DDTEST (SUB3, 2)
  there are no sublists to invoke DDTEST on.
  compares statements 6 and 6 at level 2.
  returns.
  compares statements 2 and 2 at level 1.
  compares statements 2 and 3 at level 1.
  compares statements 2 and 5 at level 1.
  compares statements 2 and 7 at level 1.
  compares statements 3 and 3 at level 1.
  compares statements 3 and 5 at level 1.
  compares statements 3 and 7 at level 1.
  compares statements 5 and 5 at level 1.
  compares statements 5 and 7 at level 1.
  compares statements 7 and 7 at level 1.
  compares statements 4 and 2 at level 1.
  compares statements 4 and 3 at level 1.
  compares statements 4 and 5 at level 1.
  compares statements 4 and 7 at level 1.
  compares statements 6 and 2 at level 1.
  compares statements 6 and 3 at level 1.
  compares statements 6 and 5 at level 1.
  compares statements 6 and 7 at level 1.
  compares statements 4 and 6 at level 1.
  returns.
  compares statement 1 with every other statement at level 0.
  returns.

Figure 9. Order of Comparisons in Sample Program.

program in this manner. First, it allows certain implicit dependences
always present in a program to be explicitly detected. An example is
the dependence that every statement has on a DO loop surrounding it.
Every statement in a DO depends on that DO at least implicitly in the
number of times the statement is executed. The fact that the dependence
is often implicit makes it very hard to detect using Banerjee's test and
the gcd test. Once the DO is converted to an assignment, however, it
can be handled within the framework of regular dependence analysis.
Second, encoding loop upper bounds makes code generation considerably
easier. Since the object of vectorization is to distribute as many DO's
around a statement as possible, DO statements are quite often syn-
thesized during code generation. These statements are either implicitly
present in statements that have been converted to parallel form, or are
explicitly present for statements that could be coded in parallel but
have no parallel equivalent. Regardless of their form, the upper bounds
for each synthetic DO must be easily obtainable at code generation.
Once the upper bounds of each loop are assigned to an easily recognized
variable, this process is greatly simplified.

6.3. Input and Output Lists

If Banerjee's test and the gcd test are to be efficiently imple-
mented, some method must be devised to quickly find the variables that a
statement uses and the variables that a statement defines. Addition-
ally, this method should include some way to locate the terms necessary
to evaluate Banerjee's test and the gcd test for a given subscript. FFC
handles these issues by having two lists of variables for every state-
ment: one to hold the input variables of the statement and one to hold
the output variables of the statement. The format of these lists allows
both rapid determination of the common variables in two statements and
the rapid location of the terms that appear in the dependence tests.

The input lists and output lists are identical both in form and in
the information contained. The information contained in the lists
includes:

(1) the symbol table index of the variable,
(2) the type of the variable (either SCALAR, ARRAY, or COMMON).

(3) the number of dimensions of the variable (nonzero only for ARRAY types).

(4) a header for a subscript list for the variable (if it is an ARRAY).

(5) a header for the next node in the list, and

(6) a header for a list of pointers that hold the exact location of the variable in the AST.

The lists are sorted by increasing symbol table index so that variables common to two lists can be quickly found by a merge procedure. Each array reference has a header for a subscript list. Nodes in subscript lists contain the number of the subscript, the linearity of the subscript, and a header for a coefficient list. Coefficient lists are sorted by increasing term number, and contain the coefficients of the corresponding term in the dependence tests.

A sample input list for the statement $X = A \times (-5 \times \&l, \&l + 2) + B$ (assumed to be contained in one DO loop) is given in Figure 10. Several things are noteworthy about the example. First, two mysterious variables $\&l$ and $\&RL$ appear in the lists. $\&l$ is the loop control variable created by STDDNRH for the DO statement controlling level 1 statements. $\&RL$ is the variable created by DDBLDSL to hold the upper bounds of the same DO statement. Second, $\&RL$ appears in the the input list to the example even though it is not an explicit input to the statement. The reason for this is to force a dependence between this statement and the assignment statement DDBLDSL created to replace the DO statement. This dependence is one that actually exists in the program, but which could not be detected in normal form by our analysis. Third, the coefficient lists are fully expanded; that is, all the terms of Banerjee's test appear in the list, even those with zero coefficients. Finally, the

---

1 The COMMON type is a catchall for variables that do not fit into the mold required by the dependence tests. Declared common variables are obviously one example. Another is an array which has its diagonal written out on each iteration of a loop. The parts of that array used cannot be conveniently represented for dependence testing.
Figure 10. Input List for \( X = A \left(-5 \times a_1, a_1 + 2\right) + B \)
header for the list of pointers into the AST does not appear in the diagram, because there is no way to graphically represent it without a diagram of the AST for the statement.

Building the input output lists is a relatively simple matter. In PFC, the lists are built by DDBLDIO, a procedure which walks the AST passing down a parameter to indicate whether variables found are to be added to the input list, the output list, or to both. The lists are maintained in sorted order as the tree is walked, so that duplicates are easily detected.

The part of the list that is hardest to build is the coefficient list. The section of DDBLDIO that does this acts like a recursive descent parser: it dives into sections of AST hunting for loop induction variables, pulling out the coefficients of those it finds. The fact that all subscripts are in standardized form simplifies the task considerably.

The algorithm for COMPARE, which uses the input output lists, is given in Figure 11.

6.4. Evaluation of Banerjee's Inequality

The procedures executed the most often in PFC are the ones that actually evaluate Banerjee's inequality and the gcd test. Since these procedures are called once for each common variable at each common nesting level in each of \( n^2 \) pairs of statements, it is vital to PFC that these routines be efficient. Even though there is a tremendous amount of preparatory work in DDBLDIO and DDBLDSL, we still felt that the bottleneck of PFC would be the dependence testing. By being very careful in the way we evaluated the tests, we achieved a much faster running time than we expected.

The gcd test is easy to evaluate rapidly. First, it determines integer solutions to the dependence equation independent of any particular region. Therefore, it only needs to be computed once for each possible level of dependence, rather than twice like Banerjee's test. Second, the gcd of the terms that typically appear in subscripts becomes 1 very quickly. Since 1 will always divide \( a_0 - b_0 \), there is no point
procedure COMPARE (S1, S2, LEVEL);
  for I + 1 to 3 do
    case I in
      begin /* 1: Check for true dependence of S2 on S1 */
        L1 + output list of S1; L2 + input list of S2;
        DEPENDENCE_OF_S2_ON_S1 + TRUE_DEPENDENCE;
        DEPENDENCE_OF_S1_ON_S2 + ANTI_DEPENDENCE
      end;
      begin /* 2: Check for true dependence of S1 on S2 */
        L1 + input list of S1; L2 + output list of S2;
        DEPENDENCE_OF_S2_ON_S1 + ANTI_DEPENDENCE;
        DEPENDENCE_OF_S1_ON_S2 + TRUE_DEPENDENCE
      end;
      begin /* 3: Check for output dependence */
        L1 + output list of S1; L2 + output list of S2;
        DEPENDENCE_OF_S2_ON_S1 + OUTPUT_DEPENDENCE;
        DEPENDENCE_OF_S1_ON_S2 + OUTPUT_DEPENDENCE
      end
    esac;
    while (not_exhausted (L1) & not_exhausted (L2)) do
      if key (L1) < key (L2) then move_down (L1)
      else if key (L2) > key (L1) then move_down (L2)
      else /* We have a match */
        for J + LEVEL to 1 step -1 do
          INDEPENDENT false;
          gcd (key (L1), key (L2), J, INDEPENDENT);
          if not INDEPENDENT then
            begin
              banerjee (key(L1), key (L2), J, INDEPENDENT) fi;
              if not INDEPENDENT then add an edge to the dependence graph
                indicating that S2 depends on S1 at level J
                with type DEPENDENCE_OF_S2_ON_S1 fi;
              if not S1 = S2 then repeat the process for S1 depending on S2 fi
            end
          od;
        move down one of the lists, watching for multiple references
          to the same array in the input list of a statement.
        fi
      od
    end
  end

Figure 11. Algorithm for COMPARE.
in continuing the test once a gcd of 1 appears; the gcd test will have to assume dependence. This one optimization alone seems to provide PFC with all the speed it requires for evaluating the gcd.

A casual examination of Banerjee's inequality

\[ \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \]

\[ \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1) - (a_k - b_k)^+(N_k - 1) - \sum_{i=k+1}^{n_2} a_i (N_i - 1) - \sum_{i=k+1}^{n_1} a_i (N_i - 1) \]

\[ \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) + (a_k - b_k)^-(N_k - 1) + \sum_{i=k+1}^{n_2} b_i (N_i - 1) + \sum_{i=k+1}^{n_1} b_i (N_i - 1) \]

does not reveal any obvious way to optimize its computation, other than to compute the middle term once and save its value for future uses. However, closer examination reveals some interesting facts about the left side of the inequality. First, every term which is superscripted by a minus or a plus is nonnegative - i.e. \((a_i - b_i)^-\) has to be greater than or equal to 0 by definition of the negative part of a number. If one assumes that the upper bounds of each loop is at least 1, then every term that is subtracted on the left hand side of the inequality is greater than or equal to 0. Similarly, every term that is added on the right hand side of the inequality is greater than or equal to 0. Thus, the inequality takes the form

\[ 0 = \text{nonnegative number} \]

\[ \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \]

\[ \leq 0 + \text{nonnegative number} \]

Therefore, at most one side of the inequality can be violated. That is, either the middle sum is less than 0, in which case only the left inequality could be false; or the middle sum is greater than 0, in which case only the right inequality could be false. If the middle sum evaluates to 0, then neither inequality can be false and we have no need to
evaluate the left or right sides. This suggests a preliminary method for performing Banerjee's test:

1. Compute (or have available) the middle term of Banerjee's inequality.

2. If the middle term is less than 0, then compute the left side of the inequality. If the left side is greater than the middle term, then the statements are independent (at the level tested); otherwise we must assume dependence.

3. If the middle term is greater than 0, then compute the right side of the inequality. If the right side is less than the middle term, then the statements are independent (at the level tested); otherwise we must assume dependence.

4. If the middle term is 0, then there is no reason to compute either side of the inequality, since it cannot possibly show independence.

The same algorithm can be used with the other form of Banerjee's inequality, if \(-b_k\) is substituted for 0.

The above procedure works well when the upper bounds of all loops involved are known constants, but unfortunately, this is not always the case. However, the procedure can be easily extended to handle general expressions in loop upper bounds. Given the assumption that all loop upper bounds are at least 1, every term subtracted from the left hand side of Banerjee's inequality must be nonnegative. In other words, the term for any level

\[-(\text{expression})(N_i-1)\]

always decreases the value of the left side of the inequality.

Since we can show independence on the left inequality only when the left side is greater than the middle, any unknown upper bound expression that has a nonzero coefficient on the left side of the inequality forces an assumption of dependence because we can make the unknown multiplying upper bound as large as we wish. On the other hand, if all the unknown upper bound expressions on the left side have zero coefficients, and if the sum of the known quantities is greater than the middle term, then the statements are independent regardless of the values of the
unknown upper bounds. These observations lead to a generalized method for evaluating Banerjee's inequality:

(1) Compute (or have available) the middle term of the inequality.

(2) If the middle term is less than 0, then compute the sum of all the known quantities of the left side of the inequality. If this sum is less than or equal to the middle term, the test cannot show independence. If the sum is greater than the middle term, then compute the coefficients of all the unknown terms. If all these coefficients are 0, then the statements are independent; otherwise, dependence must be assumed.

(3) If the middle term is greater than 0, compute the sum of all the known quantities on the right side of the inequality. If the sum is greater than or equal to the middle term, the test cannot show independence. If the sum is less than the middle term, then compute the coefficients of all the unknown terms. If these are all 0, the statements are independent; otherwise, dependence must be assumed.

(4) If the middle term is 0, dependence must be assumed regardless of the values of the other sides of the inequality.

Finally, note that when there is only one symbolic upper bound which has a nonzero coefficient in the case where dependence must be assumed, the threshold (or loop size for which that dependence is broken) can be computed from the known information. Thus, even though complete vectorization is not possible in that case, some partial vectorization may be feasible.

6.5. Overall Design

The previous sections have presented the details of the important phases of the dependence analysis process. This section will try to piece them all together and provide insight into the major design decisions of PFC.

Probably the most notable aspect of the previous sections is the use of linked lists. Lists are used in induction variable substitution,
in the algorithm that generates the statement comparisons, and as a method for storing the inputs and outputs of a statement. This extensive use of linked lists was recognized early in the design of PFC. Rather than going through the arduous task of debugging the linked list operations for every set of linked lists in PFC, we decided to implement a set of general linked list utilities. These utilities, known as POLYLIST, are capable of creating and manipulating general linked lists. Nodes of POLYLIST lists have two programmer accessible fields: a key field and a value field. A programmer can specify any data type for these fields that he desires (hence the name POLYMorphic LIST utilities). All references to lists are in a high level notation similar to that used throughout this paper. Several examples of POLYLIST operations are given below. Their functions should be obvious from their names.

```
declare_list_type (INPUT_TYPE) keytype (FIXED BINARY)
  valtype (LIKE INPUT_STRUCT);  
declare_list (INPUT_LIST) type (INPUT_TYPE) space(IO_SPACE);
move_to_head (INPUT_LIST);
new_head (INPUT_LIST);
move_down (INPUT_LIST);
find_in_list (INPUT_LIST) key(SYM_INDEX) flag(FOUND);
while (not_exhausted (INPUT_LIST)) do
```

In order to provide truly polymorphic utilities, POLYLIST was written as PL/I statement macros that expand to in-line code. Thus POLYLIST not only provides data abstraction, it also executes rapidly at run time.

The concept of extended blocks introduced in the section on subscript normalization plays an important role in determining the sections of the program on which the dependence analysis will be called. Subscript normalization, dependence analysis, and parallel code generation are all called only on the extended blocks of a program. These are the only sections where parallel code may be generated using only data dependence; other sections require a nontrivial form of control flow analysis as well. This reduces the complexity of the analysis tremendously; the \( N^2 \) time required by DUTEST is reduced to the sum of squares
of numbers smaller than \( N \) without a significant reduction in the amount of vectorization possible.

A brief outline of the entire dependence testing phase as it is presently implemented is given below:

(1) Partition the program into extended blocks.

(2) Call STDDRM and STIVSUB to perform subscript normalization and induction variable substitution on every extended block in the program.

(3) Call DDLDIO to build input output lists for the entire program. This is done to provide information for scalar data flow analysis as well as for the dependence analysis.

(4) For each extended block, call DDLDSSL to build a statement list, then call DDTTEST to perform the actual dependence analysis.

At this point the dependence graph exists for the extended blocks, and all the tools are ready to transform the program to parallel form.
7. Parallel Code Generation

The parallel code generation routine was outlined in Section 4. The purpose of this section is to show how the concept of layered dependence makes the generation of parallel code easier. In the process, we shall show how the topological sort and strongly-connected region finding routines are incorporated into the overall process.

Throughout the discussion in this section, we shall refer to a somewhat contrived example that illustrates a number of important points. The original Fortran for that example is shown in Figure 12.

```
DO 30 I = 1, 100
   X(I) = Y(I) + 10
   DO 20 J = 1, 100
      B(J) = A(J, N)
      DO 10 K = 1, 50
         A(J+1, K) = B(J) + C(J, K)
      10 CONTINUE
   20 CONTINUE
30 CONTINUE
```

Figure 12. Parallel Code Generation Example.

The dependence graph for this example (in which only dependences involving the numbered statements are displayed) is shown in Figure 13.

Let us track the code generation process through this example to illustrate a few points. At the top level, statements (2), (3), and (4) are involved in a large recurrence, so that code generation will be called recursively for these statements one level down. Parallel code will be generated for statement (1) after the loop for the first three.

```
DO 30 I = 1, 100
   code for (2), (3), (4) generated at lower levels
30 CONTINUE
```

At the next level down, the output dependences of (1), (2), and (3) on themselves, which occur because the array being assigned into does not have enough subscripts for all the surrounding loops, disappear.
Also, the antidependence of (3) on (4), due to the possibility that $A(J+1,K)$ is the same as $A(J+1,N)$ on a successive iteration of the outer loop, is broken. This leaves the dependence graph shown in Figure 14. Statements (2) and (3) still form a recurrence, but code can now be generated for statement (4).

```
DO 30 I = 1, 100
DO 20 J = 1, 100
  code for (2), (3) generated at lower levels
20    CONTINUE
      Y(I+1:I+100) = A(2:101,N)
(4)
30    CONTINUE
      X(I:100) = Y(I:100) + 10
(1)
```

The recurrence involving statements (2) and (3) will be broken at the next level down and the final code shown in Figure 15 will result.
Figure 14. Dependence graph at level 2.

```
DO 30 I = 1, 100
  DO 20 J = 1,100
    B(J) = A(J,N)
  A(J+1,1:100) = B(J) + C(J,1:100)
  CONTINUE
  Y(I+1:1+100) = A(2:101,N)
  CONTINUE
X(1:100) = Y(1:100) + 10
```

Figure 15. Final code for Figure 12.

This example is pleasing because it generates vector statements at three different levels and because it illustrates, in the case of statement (2), how parallel code generation can also serve as scalar code generation. This happens when we generate code that is parallel in 0 dimensions.

There are two aspects of the code generator that are worthy of further note. First the strongly-connected region finder is a direct adaptation of Tarjan's depth-first search method [Tarj 72]. The time required by this method is linear in the number of statements and dependence edges remaining in the graph.
Second, the statements in the graph are ordered by an adaptation of the topological sort [Knut 73] which computes the derived dependence relation between π-blocks on the fly. The basic idea is simple. In the first pass through the edges of the dependence graph, a distinction is made between edges within a π-block, which are ignored, and those between π-blocks, which are used to set up the dependence for topological sort and then deleted from the graph (this is fine since further processing will be at a lower level and hence within a region).

The algorithm is of interest because it illustrates the structure of the dependence graph as well as the nature of topological sort. The structure of the dependence graph is shown in Figure 16. Each statement contains, in addition to the standard useful information, pointers to the headers of lists of dependence edges into and out of the statement (along with a count of the edges in the list). Each edge is a member of two doubly-linked lists, one of edges to the same sink and the other of edges from the same source.

---

**Node Structure**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_DEPEND_ON</td>
<td>(list of edges in)</td>
</tr>
<tr>
<td>#I_DEPEND_ON</td>
<td></td>
</tr>
<tr>
<td>DEPENDS_ON_ME</td>
<td>(list of edges out)</td>
</tr>
<tr>
<td>#DEPENDS_ON_ME</td>
<td></td>
</tr>
<tr>
<td>PIBLOCK_NAME</td>
<td>(containing piblock)</td>
</tr>
</tbody>
</table>

**Edge Structure**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>(source stmt number)</td>
</tr>
<tr>
<td>TO</td>
<td>(sink stmt number)</td>
</tr>
<tr>
<td>TYPE</td>
<td>(anti, true, output)</td>
</tr>
<tr>
<td>LEVEL</td>
<td>(of carrier loop)</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td></td>
</tr>
<tr>
<td>NEXT_FROM</td>
<td>(next edge from same source)</td>
</tr>
<tr>
<td>PREV_FROM</td>
<td>(prev edge from same source)</td>
</tr>
<tr>
<td>NEXT_TO</td>
<td>(next edge to same sink)</td>
</tr>
<tr>
<td>PREV_TO</td>
<td>(prev edge to same sink)</td>
</tr>
</tbody>
</table>

---

Figure 16. Structure of the dependence graph.
As in all topological sorts, the algorithm makes use of two arrays indexed by \( \pi \)-block number. \texttt{SUCCESSORS}[I] is a list of \( \pi \)-block numbers of blocks which depend on \( \pi \)-block I. \texttt{NPRED}[I] contains the number of \( \pi \)-blocks not yet added to the final order upon which \( \pi \)-block I depends. The algorithm is displayed in Figure 17.

One wrinkle in the implementation is not shown. The usual linear time bound (linear in the number of vertices and edges) of topological sort had to be sacrificed for cosmetic reasons. When presented with a group of statements having no dependences among them, the version of PFC with a standard topological sort tended to rearrange those statements into random order. This unnecessary code motion causes confusion for the user attempting to read a translated program, so we eliminated it by implementing \texttt{NO_PRED} as a priority queue (heap), to produce a stable topological sort that operates in \( O(n \log n + e) \) time, where \( n \) is the number of \( \pi \)-blocks and \( e \) is the number of dependence edges.

8. Auxiliary Transformations

Before giving up and generating serial code for a statement, the code generator will try a number of auxiliary transformations to expose more parallelism. We will discuss several of these in this section.

8.1. Recurrence Breaking

A number of recurrences, particularly those based upon output dependences and antidependences, can be broken by clever renaming or the introduction of array temporaries. These transformations are described in a previous paper [Ken 80].

In addition, a number of such recurrences are familiar patterns for which special language constructs may exist. For example, consider the following loop.

\[
\begin{align*}
S &= 0.0 \\
\text{DO 10 I = 1, N} \\
S &= S + X(I) \\
10 &\text{ CONTINUE}
\end{align*}
\]

This can be easily recognized to be a sum reduction, and replaced by a call to an efficient library function.
/* set up the auxiliary data structures */
for I = 1 TO NPIELOCKS do
  make the list SUCCESSORS[I] empty;
  NPREDs[I] = 0
od;

/* scan the edges and initialize successor lists */
/* and predecessor counts */
for I = 1 to NPIELOCKS do
  for each statement S in PIELOCK[I] do
    for each dependence edge E out of S do
      if SINK[E] is in PIELOCK[I] then
        ignore the edge, it is within a region
      else
        let J be the number of the \( \pi \)-block that
        contains the sink;
        add J to the list SUCCESSORS[I];
        NPREDs[J] = NPREDs[J] + 1;
        delete the edge E from the graph
      fi
    od
  od

/* initialize the set of \( \pi \)-blocks with no predecessors */
make NO_PREDs empty;
for I = 1 to NPIELOCKS do
  if NPREDs[I] = 0 then add I to the set NO_PREDs fi
od;

/* add the nodes to the output list one at a time */
/* reducing the predecessor count of each successor */
while (NO_PREDs is not empty) do
  NEXT_BLOCK = some element in the set NO_PREDs;
  remove NEXT_BLOCK from NO_PREDs;
  add NEXT_BLOCK to the output list;
  for each block I in SUCCESSORS[THIS_BLOCK] do
    NPREDs[I] = NPREDs[I] - 1;
    if NPREDs[I] = 0
      then add block I to the set NO_PREDs fi
  od
od

Figure 17. The modified topological sort.
8.2. Loop Interchange

Many times a recurrence at the deepest level blocks the vectorization of two loops. Under certain conditions, the two loops may be interchanged to yield a new inner loop that can be vectorized. To illustrate the power of this technique, consider the standard algorithm for matrix multiplication on a scalar machine.

\[
\text{DO 100 } J = 1, 100 \\
\text{DO 90 } I = 1, 100 \\
\quad C(I, J) = 0.0 \\
\text{DO 80 } K = 1, 100 \\
\quad C(I, J) = C(I, J) + A(I, K) \times B(K, J) \\
\text{80 CONTINUE} \\
\text{90 CONTINUE} \\
\text{100 CONTINUE}
\]

After loop distribution and interchange of the innermost two loops, this becomes

\[
\text{DO 10 } J = 1, 100 \\
\text{DO 5 } I = 1, 100 \\
\quad C(I, J) = 0.0 \\
\text{5 CONTINUE} \\
\text{10 CONTINUE} \\
\text{DO 100 } J = 1, 100 \\
\text{DO 80 } K = 1, 100 \\
\quad \text{DO 90 } I = 1, 100 \\
\quad \quad C(I, J) = C(I, J) + A(I, K) \times B(K, J) \\
\text{90 CONTINUE} \\
\text{80 CONTINUE} \\
\text{100 CONTINUE}
\]

The parallel code generator then produces:

\[
C(1:100, 1:100) = 0.0 \\
\text{DO 100 } J = 1, 100 \\
\text{DO 80 } K = 1, 100 \\
\quad C(1:100, J) = C(1:100, J) + A(1:100, K) \times B(K, J) \\
\text{80 CONTINUE} \\
\text{100 CONTINUE}
\]

The final code can be recognized as the optimal method for matrix multiplication on a vector machine with vector registers. Since the current section of \( C \) is kept in a register until all computations involving it are done, and only then put away, a vector register machine does one vector load for every two operations -- a nice balance. The Cray-1 achieves 125 megaflops on this loop [Russ 78]. The fact that optimal
vector code can be produced by a well understood transformation of the scalar code is very satisfying.

8.3. If Conversion

Most parallel machines have some method for conditionally executing an operation on elements of an array. This is usually accomplished by using some sort of bit mask to turn off execution of the operation in certain subscript positions. Fortran 8x provides the WHERE statement to exploit such hardware, and PFC makes an aggressive attempt to eliminate conditional transfers of control by replacing them, whenever possible, with WHERE statements.

If dependences permit, conditional assignments within loops, like the one below:

```
   DO 100 I = 1, 100
       IF (A(I).GT.0) B(I) = B(I) + A(I)
   100 CONTINUE
```

can be directly converted to WHERE statements:

```
   WHERE (A(1:100).GT.0) B(1:100) = B(1:100) + A(1:100)
```

To take advantage of this possibility, PFC uses a preliminary transformation called if conversion which attempts to transform conditional transfers of control to conditional assignments.

```
   DO 100 I = 1, 100
       IF (A(I).LT.0) GO TO 100
           B(I) = B(I) + A(I)
   100 CONTINUE
```

becomes

```
   DO 100 I = 1, 100
       M(I) = A(I).LT.0
       IF (.NOT.M(I)) B(I) = B(I) + A(I)
   100 CONTINUE
```

which can be directly converted to vector code.

This approach is not without its complexities. The conditions controlling execution of statements tend to become complex, and require sophisticated simplification, and the handling of jumps out of loops is tricky. However, the transformation is well worth the effort, because without it, loops with conditionals might well be left in sequential
9. Status of the Implementation

As we pointed out in the introduction, the original implementation was based upon the Parafrase compiler developed by Kuck and others at the University of Illinois [KLWL 80, KLKP 81]. We received a version of Parafrase in December of 1978, and made a number of improvements to it including the introduction of an if conversion phase and an integer expression simplifier.

Because our improved Parafrase was too slow and used an unsuitable intermediate program form, we abandoned it in January 1981, replacing it with a front end, written at Rice, that produces abstract syntax trees as an intermediate form. We then incorporated the new standardization, dependence testing, and parallel code generation phases described here.

PFC was a great improvement over our version of Parafrase (although that program has now been substantially improved at Illinois). Particularly startling was the reduction in running time. A 300-statement Fortran program took 10 minutes of IBM 370/168 CPU time using the old translator, while PFC now takes approximately 20 seconds on the same input.

PFC currently consists of about 25,000 lines of PL/I, of which about a third are comments. This figure is a bit deceiving however, since the use of macro packages like POLYLIST effected a significant compaction of the code.

A number of features described here are not yet implemented.

(1) The global data flow analysis phase, which will permit global constant folding and dead code elimination, is just now being completed.

(2) Although if conversion was implemented in the old version, we have not yet incorporated it into the new PFC.

(3) Only a few simple recurrence breaking transformations are currently used by PFC. More are planned for the near future.
(4) Loop interchange, an important part of our plans, has not yet been completed.

(5) The current pretty printer needs to be made more intelligent and a routine that chooses names for the translator-created variables needs to be written.

PFC is implemented under the IBM MV5 operating system at Rice and under VM 370 at IBM Thomas J. Watson Research Center. We are also well along on a project to produce a companion Fortran 8x compiler.

By producing an ambitious translation of sequential Fortran to parallel Fortran at surprisingly low cost, PFC has demonstrated that a significant level of vectorization can be conveniently provided by an automatic tool. Of course in specific instances, the human programmer will always be more clever than programs like PFC. However, automatic translators may eventually free the human from much of the tedious work of converting a long program and thus permit him to concentrate his creativity on the frequently executed regions of such a program.
References


