CHAPTER 2

Dependence, Parallelization and Vectorization

2.1 Introduction

One of the most pressing problems faced by the authors of the first FORTRAN compiler was making the compiled code fast enough to satisfy its users. In the words of John Backus, reflecting in 1978 on the motivation for the FORTRAN I compiler design [\cite Back^78]:

> It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger. ..To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread

> use of languages like FORTRAN would have been seriously delayed.
>
> In fact, I believe that we are in a similar, but unrecognized, situation today: in spite of all the fuss that has bee made over myriad language details, current conventional languages are still very weak programming aids, and far more powerful languages would be in use today if anyone had found a way to make them run with adequate efficiency.

Obviously, optimization has been an important function of compilers from their very inception. As a result, optimizing technology has matured to a sophisticated state. However, most of the technology has been developed with the target of a scalar machine. On such machines, the principal optimization is reducing the cost of address calculations by eliminating scalar integer expressions. Parallelism introduces much more complexity into the optimization process. With parallelism, the principle optimization becomes finding parallelism in a sequential code and tailoring that parallelism to the target machine (including its memory hierarchy). The principal strategy for seeking out useful parallelism is to look for a *data decomposition* in which parallel tasks perform similar operations on different elements of the data arrays. In FORTRAN, this strategy naturally becomes examining different iterations of DO-loops for parallelism. Data decomposition is effective on scientific problems because it is *scalable*—as the problem size increases, so does the available parallelism

Given the focus in FORTRAN on DO-loop iterations, an advanced compiler must be able to determine whether Bernstein's conditions are satisfied for every pair of iterations. Examining loop iterations is complicated enough; adding to the complexity is the fact that most code within a data-parallel loop will reference subscripted array variables. As a result, the compiler must have the ability to analyze such references to determine whether two different iterations access the same memory location. Given the fact that one array contains multiple elements, it is not always obvious when two iterations or statements refer to different elements of the same array. Because of this added complexity, the simple definition of dependence presented in Chapter 1, while elegant and easy to understand, is ineffective in the presence of loops and array references.

This chapter elaborates and refines the simple definition of dependence presented in Chapter 1 to associate dependences with specific loops, thereby permitting optimization on a loop-by-loop basis. The power of these refinements is illustrated in applications to automatic parallelization and vectorization. The chapter concludes with an advanced vectorization algorithm that is at the heart of several commercial vectorizers.

2.2 Dependence and its Properties

Section 1.3 on page 33 outlined the fundamental approach taken in this book to uncovering parallelism. The basis of this approach is a *dependence relation* or *dependence graph*, that includes, for a given sequential program, all statement-to-statement execution orderings that *must* be preserved if the meaning of the program is to remain the same. Each pair of statements in the graph is called a *dependence*. Given a correct dependence graph for the program, any ordering-based optimization that does not change the dependences of a program will be guaranteed not to change the results of the program.

Dependences can arise from two general sources. First, one obvious requirement for preserving the meaning of a program is executing the statements in an order that preserves the original load-store order. If {S_1, S_2, ..., S_n} is the sequence of statements that access a given memory location M in the sequential program, a correctly transformed program should have the property that, if S_i accesses M before S_j in the original program and one of the two accesses is a write, then S_i must access M before S_j in the transformed program. The dependences that arise from these constraints are called *data dependences*. Revisiting the example from Chapter 1:

```
S1  PI = 3.14
S2  R = 5.0
S3  AREA = PI * R ** 2
```

statement S_3 cannot be correctly executed before either S_1 or S_2. If S_3 were to be executed before either S_1 or S_2, the value used for the variable PI or R would be one that is assigned earlier in the program. This would undoubtedly cause S_3 to compute an incorrect value for AREA. As a result, S_3 has data dependences upon statements S_1 and S_2. There is no execution constraint between S_1 and S_2 however; the execution order S_2, S_1, S_3 will produce exactly the same value for PI as the execution order S_1, S_2, S_3.

The other effect that gives rise to dependences is control flow. For example, in the following code

```
S1  IF (T .NE. 0.0) GOTO S3
```

```
S2      A = A / T
S3   CONTINUE
```

statement S_2 cannot be executed before S_1 in a correctly transformed program, because the execution of S_2 is conditional upon the execution of the branch in S_1. Executing S_2 before S_1 could cause a divide-by-zero exception that would be impossible in the original version. A dependence that arises because of the control flow is called a *control dependence*.

Although both data and control dependences must be considered when correctly parallelizing a program, the next few chapters will concentrate exclusively on data dependences. This is because data dependences are more numerous and they demonstrate most of the important principles. Furthermore, Chapter 7 will show how all control dependences can be converted to data dependences by a technique known as "IF-conversion." Thus, all of the algorithms based exclusively on data dependences will immediately be generalized. Later in that same chapter, new algorithms will be introduced for dealing directly with control dependences.

Formalizing the previous discussion produces the following definition of data dependence.

> **Definition 2.1.** There is a *data dependence* from statement S_1 to statement S_2 (statement S_2 *depends on* statement S_1) if
>
> 1) both statements access the same memory location and at least one of them stores into it and
>
> 2) there is a feasible run-time execution path from S_1 to S_2.

The next few sections will discuss various properties by which dependences can be classified. These properties are important for understanding the algorithms that will be presented later in the book.

2.2.1 Load-Store Classification

Expressed in terms of load-store order, there are three ways that a dependence can arise in a program.

1. The first statement stores into a location that is later read by the second statement:
```
S1   X = ...
S2   ... = X
```
   In this case the dependence is needed to ensure that the second statement receives the value computed by the first. Execution constraints

of this form are known as *true dependences* or *flow dependences* and are denoted $S_1 \delta\, S_2$ (read S_2 depends on S_1). Note that the convention in graphical displays of dependence is to depict the edge as flowing from the statement that executes first to the one that follows in execution order.

2. The first statement reads from a location into which the second statement later stores:
```
S1   ... = X
S2   X = ...
```
Here the dependence is needed to prevent the interchange of S_1 and S_2, which could lead to S_1 incorrectly using the value computed by S_2. In essence, this dependence is present to prevent a program transformation from introducing a new true dependence that did not exist in the original program. Hence, it is called an *antidependence.* An antidependence such as this one is denoted $S_1 \delta^{-1} S_2$.

3. Both statements write into the same location
```
S1   X = ...
S2   X = ...
```
A dependence is needed to prevent a third statement later in flow from reading the wrong value. For example, in the code fragment:
```
S1   X = 1
S2   ...
S3   X = 2
S4   W = X * Y
```
statement S_3 should not be allowed to move before statement S_1 lest Y be incorrectly multiplied by 1, rather than 2, in S_4. This type of dependence is called an *output dependence* and is denoted $S_1 \delta^{o} S_2$.

2.2.2 Loop Iterations

Extending the concept of dependence to loops requires some way to parameterize statements by the loop iterations in which they are executed. For example, in the simple loop nest:

```
     DO I = 1, N
S1      A(I+1) = A(I) + B(I)
     ENDDO
```

the statement S_1 on any loop iteration depends on the instance of itself from the previous iteration. The statement "S_1 depends upon itself." is true in this case, but is not precise. For instance, what if the loop were changed to

```
      DO I = 1, N
S1       A(I+2) = A(I) + B(I)
      ENDDO
```

Even though S_1 still depends on itself, the instance has changed from the previous iteration to the one two iterations back. Much more complex examples can be constructed for multiple loops.

Precise characterization of dependences in loops requires that statements be parameterized by some representation of the loop iteration in which the statement occurs. One convenient representation of specific loop iterations is an *iteration vector,* which contains one entry for each loop surrounding the statement. The first entry in the vector corresponds to the outmost loop around the statement; the second entry corresponds to the next inner loop; and so on. A statement parameterized by a specific execution vector denotes the instance of that statement executed when the loop induction variables have the values in the execution vector. For example, one iteration vector for statement S_1 in

```
DO I = 1, 2
   DO J = 3, 4
      S1
   ENDDO
ENDDO
```

is the vector (2,3), which represents the execution of S_1 that occurs when I is equal to 2 and J is equal to 3. The set of all possible iteration vectors for a statement is an *iteration space*. The iteration space of S_1 in the above example is {(1,3), (1,4), (2,3), (2,4)}.

Because of the importance of execution order to dependence, iteration vectors need an ordering that corresponds to the execution order of their loops. Assuming the notation that $\mathbf{i}$ is a vector, $\mathbf{i}_k$ is the kth element of the vector $\mathbf{i}$, and $\mathbf{i}[1:k]$ is a k-vector consisting of the leftmost k elements of $\mathbf{i}$, the *lexicographic order* on vectors of length n is defined as follows:

$\mathbf{i} < \mathbf{j}$ (iteration $\mathbf{i}$ *precedes* iteration $\mathbf{j}$) if and only if

1) $\mathbf{i}[1:n\text{-}1] < \mathbf{j}[1:n\text{-}1]$ or

2) $\mathbf{i}[1:n\text{-}1] = \mathbf{j}[1:n\text{-}1]$ and $\mathbf{i}_n < \mathbf{j}_n$

In other words, an iteration vector $\mathbf{i}$ precedes another iteration vector $\mathbf{j}$ if and only if any statement executed on the iteration described by $\mathbf{i}$ is executed before any statement on the iteration described by **j**. The relations $\leq$, $>$, and $\geq$ can also be defined on iteration vectors by the natural extensions.

2.2.3 Dependence and Reordering Transformations

Most optimizations discussed in this book are "reordering transformations."

> **Definition 2.2.** A *reordering transformation* is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statements.

Since a reordering transformation does not delete any executions, any two executions that reference a common memory element *before* a reordering transformation will also reference the same memory element *after* that transformation. Hence, if there is a dependence between the statements before the transformation, there will also be one afterwards. Note, however, that the transformation may *reverse* the order in which the statements reference the common memory location—thereby reversing the dependence (i.e. $S_1\ \delta\ S_2$ before the transformation and $S_2\ \delta\ S_1$ after the transformation). A reordering transformation *preserves* a dependence (and thereby the semantics of the code) if it preserves the relative execution order of the source and sink of that dependence.

We are now ready to prove the central result in the theory of dependence.

> **Theorem 2.1. Fundamental Theorem of Dependence.** Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Proof. Before proving this theorem, it is necessary to define what "preserve the meaning" of a program means. Since this theorem deals with imperative languages, in which each statement reads from and stores into memory, meaning is most logically defined in terms of the *state* of the computation. If the computation leaves the same values in memory (or if each output instruction reads the same values) then the meaning is preserved. Hence, if each statement in the reordered program sees the same input values it saw in the original program and if the relative store orders are preserved (which is guaranteed by preservation of output dependences), then the meaning is preserved, since each statement will write the same value to the same location in memory.

Let us begin with loop-free programs that have no conditional statements in them. Let $\{S_1, S_2, \ldots, S_n\}$ be the execution order in the original program and $\{i_1, i_2, \ldots, i_n\}$ be the order of execution of statements in the reordered program, where $\{i_1, i_2, \ldots, i_n\}$ is a permutation of $\{1,2, \ldots, n\}$ corresponding to the reordering of the statements. Assume that dependences are preserved, but that the meaning changes. Let S_k be the first statement in the new order that sees a wrong value in one of its inputs. Since all the statements that have executed *before* S_k produce the same value they did in the original program, there are only three ways that S_k can get the wrong answer:

1. Another statement S_m, which executed and stored its result in a memory location later read by S_k in the original order, now stores into the same location after it is read by S_k. This would mean that the reordering violated a true dependence (that of $S_m \; \delta \; S_k$), contrary to assumption.
2. Another statement S_m, which wrote into a location previously read by S_k in the original order, now writes before S_k reads. This would mean that the reordering fails to preserve an antidependence (that of $S_k \; \delta^{-1} \; S_m$), contrary to assumption.
3. Two previous statements that both write into a location read by S_k have their order of execution reversed by the reordering, causing the wrong value to be left in that location. This would mean that the reordering fails to preserve an output dependences, contrary to assumption.

Since this exhausts the ways that S_k can get the wrong value, the result is proved by contradiction.

Loops. To extend this result to loops, simply note that the statement executions in the list can be viewed as statement instantiations that are indexed by the iteration vector. Then the same argument applies, providing that the correct loop iteration count is computed *before* any statement in the loop body is executed. This is necessary to ensure that the collection of statement instantiations that are executed is preserved. The normal convention for enforcing this restriction is to have a control dependence from a loop header to each statement in the body of the loop.

Programs with conditionals. Because we do not have a clean definition of what a reordering transformation is with respect to conditionals, we will assume for now that a conditional is a single macro statement (i.e., an if-then-else statement) that is treated as a unit by reordering transformations. The result will be extended to more general transformations in Chapter 7. With that single limitation, the theorem is proved. QED.

There are two immediate corollaries to this theorem.

> **Corollary 2.1. Bernstein's Conditions**. Two non-intersecting program regions R_1 and R_2 can be run in parallel without any synchronization if none of the following conditions hold:
>
> 1. R_1 writes into a location that R_2 reads.
> 2. R_1 reads from a location that R_2 writes.
> 3. R_1 and R_2 write into the same location.

Proof. We assume that the original sequential program contains no superfluous code, so that given a pair of statements S_1 from R_1 and S_2 from R_2, there is an execution path from S_1 to S_2 or vice versa. Since none of the three conditions hold, there cannot be a dependence between any statement in one region and any statement in the other. Hence we can reorder the execution of these statements relative to one another without changing the meaning of the program. In particular, we can run R_1 and R_2 in parallel. QED.

Second is a key result in the theory of automatic parallelization.

> **Corollary 2.2. Fundamental Theorem of Parallelization**. Two groups of statements G_1 and G_2 in a loop-free program can be safely executed in parallel if there is no dependence from a statement in one group to a statement in the other.

Proof. A direct consequence of the Bernstein conditions—there exists no dependence if none of the conditions holds. QED.

Unfortunately Corollary 2.2 is not very helpful, since loop-free programs rarely have enough computation to be interesting. Loops are the more interesting construct, but talking about dependences in loops requires a notation tying them to different loop iterations, just as iteration vectors tied statements to loop iterations. That notation will be the subject of the next two sections.

2.2.4 Distance and Direction Vectors

The lexicographic order of two iteration vectors can be succinctly summarized using *distance vectors* and *direction vectors* [\cite Wolf^78].

> **Definition 2.3.** If two iteration vectors **i** and **j** represent executions of statements that are contained in *n* common loops, then the

distance vector $\mathbf{d}(\mathbf{i}, \mathbf{j})$ is defined as a vector of length n such that $\mathbf{d}(\mathbf{i}, \mathbf{j})_k = \mathbf{j}_k - \mathbf{i}_k$.

Based on this definition, we have the following

Lemma 2.1. $\mathbf{i} < \mathbf{j}$ if and only if $\mathbf{d}(\mathbf{i},\mathbf{j}) > \mathbf{0}$.

Distance vectors can be associated with dependences according to the following rule.

Definition 2.4. Suppose that there is a dependence from statement S_1 on iteration $\mathbf{i}$ of a loop nest and statement S_2 on iteration $\mathbf{j}$, then the *dependence distance vector* is $\mathbf{d}(\mathbf{i},\mathbf{j})$.

Difference vectors are mapped into direction vectors by the following:

Definition 2.5. If two iteration vectors $\mathbf{i}$ and $\mathbf{j}$ represent executions of statements that are contained in n common loops, then the direction vector $\mathbf{D}(\mathbf{i}, \mathbf{j})$ is defined a a vector of length n such that

$$\mathbf{D}(\mathbf{i}, \mathbf{j})_k = \begin{cases} \text{"<" if } \mathbf{d}(\mathbf{i}, \mathbf{j})_k > 0 \\ \text{"=" if } \mathbf{d}(\mathbf{i}, \mathbf{j})_k = 0 \\ \text{">" if } \mathbf{d}(\mathbf{i}, \mathbf{j})_k < 0 \end{cases}$$

One convenient mechanism for remembering the entries in direction vectors is to treat "<" and ">" as arrows. With that treatment, the arrows always point to the iteration vector that occurs first.

Direction vectors can be related to dependences just as distance vectors can.

Definition 2.6. Suppose that there is a dependence from statement S_1 on iteration $\mathbf{i}$ of a loop nest and statement S_2 on iteration $\mathbf{j}$, then the *dependence direction vector* is $\mathbf{D}(\mathbf{i},\mathbf{j})$.

The direction vector for a dependence relates the iteration vector at the source of a dependence to the iteration vector at the sink. For example, in the loop

```
      DO I = 1, N
         DO J = 1, M
            DO K = 1, L
S1             A(I+1,J,K-1) = A(I,J,K) + 10
            ENDDO
         ENDDO
```

```
ENDDO
```

statement S_1 has a true dependence on itself with direction vector (<,=,>), meaning that the outermost loop index at the source is less than the index at the sink, the middle loop index is equal at source and sink, and the innermost loop index is larger at the sink than at the source. Note that a dependence cannot exist if it has a direction vector whose leftmost non-"=" component is not "<" because of the following lemma which is immediate from the definition of direction vectors and Lemma 2.1

> **Lemma 2.2. i < j** if and only if the leftmost non-"=" component of **D**(**i, j**) is "<".

The implication is that the leftmost non-"=" component of a dependence direction vector must be "<" because otherwise the source would be executed *after* the sink, which is impossible.

The reason for limiting direction vectors and difference vectors to common loops is that these loops are the only ones that affect the relative execution order of statements. If the first n components of two iteration vectors are equal, execution order is determined by textual position, regardless of any remaining components of the vectors.

Direction vectors can be used as a basis for understanding loop reordering transformations because they summarize the relationship between the index vectors at the source and sink of a dependence. Chapter 5 will show how it is possible to determine the effect of various transformations on the direction vectors for dependences in the loop nest affected by the transformation. The following theorem explains how direction vectors can be used to test for legality of a transformation.

> **Theorem 2.2.** Let T be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is legal if none of the direction vectors for dependences with source and sink in the nest has a leftmost non- "=" component that is ">" after the transformation.

Proof. The theorem follows directly from Lemma 2.2.

2.2.5 Loop-carried and Loop-independent Dependences

The theory of data dependence described so far imposes two requirements that must be met for a statement S_2 to be data dependent on statement S_1:

1. There must exist a possible execution path such that statements S_1 and S_2 both reference the same memory location *M*.
2. The execution of S_1 that references *M* occurs before the execution of S_2 that references *M*.

In order for S_2 to depend upon S_1, it is necessary for some execution of S_1 to reference a memory location (as a store, if true dependence is considered) that is later referenced by an execution of S_2 (as a use, for true dependence). There are two possible ways that this pattern can occur:

1. S_1 can reference the common location on one iteration of a loop; on a subsequent iteration S_2 can reference the same location.
2. S_1 and S_2 can both reference the common location on the same loop iteration, but with S_1 preceding S_2 during execution of the loop iteration.

The first case is an example of *loop-carried dependence*, since the dependence exists only when the loop is iterated. The second case is an example of *loop-independent dependence*, since the dependence exists because of the position of the code within the loops. The following sections detail these types of dependence.

2.2.5.1 Loop-Carried Dependence

A loop-carried dependence arises because of the iteration of loops. The following Fortran segment demonstrates this idea:

```
      DO I=1, N
S1       A(I+1) = F(I)
S2       F(I+1) = A(I)
      ENDDO
```

On every iteration of the `I` loop other than the first, S_2 uses a value of `A` that was computed on the previous iteration by S_1; hence, statement S_2 has a true dependence on statement S_1. Likewise, statement S_1 uses a value of `F` computed by statement S_2 on the previous iteration (except for the first), and truly depends on statement S_2. Both of these dependences are carried by the loop. If any particular iteration of the loop is chosen and executed alone, no dependence exists, reinforcing the close relationship between the loop and the dependence.

Formally, a loop-carried dependence may be defined by the following:

Definition 2.7. Statement S_2 has a *loop-carried dependence* on statement S_1 if and only if S_1 references location *M* on iteration

i, S_2 references *M* on iteration **j** and **d**(**i**,**j**) > **0** (that is, **D**(**i**,**j**) contains a "<" as leftmost non "=" component).

The appearance of a nonzero component in **d(i,j)** guarantees that the corresponding loop iterates at least once between the common references—hence the name loop-carried dependence.

An important property of loop-carried dependence is the *level* of a dependence.

Definition 2.8. The *level* of a loop-carried dependence is the index of the leftmost non-"=" of **D(i,j)** for the dependence.

In other words, the level of the dependence is the nesting level of the outermost loop index that varies between the source and sink, where the outermost loop is taken to be at nesting level 1. The level of all the dependences in the previous example is 1, since **D(i,j)** is (<) for every dependence. The level of the dependence in

```
      DO I = 1, 10
         DO J = 1, 10
            DO K = 1, 10
S1             A(I,J,K+1) = A(I,J,K)
            ENDDO
         ENDDO
      ENDDO
```

is 3 because **D**(**i**,**j**) is (=,=,<). Note that the dependence in this case is actually a collection of dependences, one for every iteration vector in the set {1:10,1:10,1:9}. In the future, we will associate a separate dependence between each pair of different subscripted array references. We will refer to all dependences for a single reference pair as being one dependence, but that dependence may have many direction vectors.

Dependence level is a useful concept for many reasons. One reason is that level very conveniently summarizes dependences. For example, the last fragment contains 900 total dependences. Since every **i**,**j** that creates a dependence has **d**(**i**,**j**) = (0,0,1), the level information conveniently characterizes all the dependences by a single quantity.

Level also forms the basis of certain program transformations. In particular, many reordering transformations can be enacted based on the level of a dependence, as the following theorem reveals:

Theorem 2.3. Any reordering transformation that does not alter loops 1 through k preserves any level-k dependence.

Proof: For any pair of iteration vectors $\mathbf{i},\mathbf{j}$ that gives rise to a level-k dependence, the leftmost "<" for $\mathbf{D}(\mathbf{i}, \mathbf{j})$ must be the kth entry. If the reordering transformation maps $\mathbf{i}$ to $\mathbf{i}'$ and $\mathbf{j}$ to $\mathbf{j}'$, then $\mathbf{i}_k' = \mathbf{i}_k$ and $\mathbf{j}_k' = \mathbf{j}_k$ because the hypothesis guarantees that loops 1 through k are unaffected by the transformation. Therefore, $\mathbf{D}(\mathbf{i}', \mathbf{j}')_k = \mathbf{D}(\mathbf{i}, \mathbf{j})_k$ and the level k dependence is not reversed by the transformation. QED.

Any transformation that preserves all dependences is known as a *valid* transformation, since it also preserves the output of the program. Theorem 2.3 establishes the validity of some powerful transformations based dependence level. For example, the first fragment in its original form

```
    DO I = 1, 10
S1     A(I+1) = F(I)
S2     F(I+1) = A(I)
    ENDDO
```

is equivalent to

```
    DO I = 1, 10
S2     F(I+1) = A(I)
S1     A(I+1) = F(I)
    ENDDO
```

since all dependences are at level 1. Additionally, Theorem 2.1 permits other transformations such as loop rearrangement and loop reversal inside the deepest dependence. For example, the following

```
    DO I = 1, 10
       DO J = 1, 10
          DO K = 1, 10
S1           A(I, J, K) = A(I-1, J-2, K-3)
          ENDDO
       ENDDO
    ENDDO
```

(which has only a level-1 dependence) is equivalent to

```
    DO I = 1, 10
       DO K = 10, 1, -1
          DO J = 1, 10
```

```
S1          A(I, J, K) = A(I-1, J-2, K-3)
          ENDDO
       ENDDO
    ENDDO
```

obtained by interchanging the J and K loops while reversing the K loop.

According to Theorem 2.1, the level of a dependence marks the innermost loop that creates the dependence. Loop interchange, discussed in Chapter 4, will strengthen that idea by showing that the level of a dependence marks the *only* loop that creates that dependence. Given the importance of dependence level, we will denote the level of a dependence via a subscript. That is, a level l dependence between S_1 and S_2 is denoted $S_1\ \delta_l\ S_2$.

2.2.5.2 Loop-Independent Dependences

In contrast to loop-carried dependence, loop-independent dependences arise as a result of relative statement position. Thus, loop-independent dependences determine the order in which code is executed within a nest of loops, while loop-carried dependences affect the order in which loops must be iterated.

Formally, a loop-independent dependence is defined by the following:

> **Definition 2.9.** Statement S_2 has a *loop-independent dependence* on statement S_1 if and only if there exists two iteration vectors **i,j** such that the following two conditions hold:
>
> 1) Statements S_1 refers to memory location M iteration **i**; S_2 refers to the same memory location M on iteration **j**; and **d**(**i**,**j**) = **0**.
>
> 2) There is a control flow path from S_1 to S_2 within the iteration.

Intuitively, Definition 2.9 states that a loop-independent dependence exists when two statements reference the same memory location within a single iteration of all their common loops. A very obvious example is

```
    DO I = 1,10
S1     A(I) = ...
S2      ...  = A(I)
    ENDDO
```

On every iteration of the I loop, statement S_2 uses the value just computed by statement S_1, thus creating a loop-independent dependence. A less obvious example is

```
    DO I = 1, 9
S1     A(I) =
S2       ... = A(10-I)
    ENDDO
```

On the fifth iteration of the loop, statement S_1stores into A(5) while statement S_2 fetches from A(5). That dependence is loop-independent. All other dependences in the segment are carried by the loop. The reason that separate iteration vectors **i** and **j** appear in the definition is illustrated by the following:

```
    DO I = 1, 10
S1     A(I) = ...
    ENDDO
    DO I = 1,10
S2       ...  = A(20-I)
    ENDDO
```

Statement S_2 uses the value of A(10) computed by statement S_1 on the tenth iteration of the first loop, creating a loop-independent dependence. No common loop is necessary for a loop-independent dependences, since they arise from statement position.

In contrast to loop-carried dependences, loop-independent dependences specify the order in which statements must be executed, as Theorem 2.4 demonstrates:

> **Theorem 2.4.** If S_2 has a loop-independent dependence on S_1, then any valid reordering transformation must have S_2 follow S_1.

Proof: By definition, S_2 and S_1 both reference some location M with a difference vector $\mathbf{d}(\mathbf{i},\mathbf{j}) = \mathbf{0}$. Assume that S_2 precedes S_1 in the transformed code. If iteration **i** is mapped by the transformation to a new iteration $\mathbf{i}'$ and iteration **j** is mapped to $\mathbf{j}'$, and if S_1 and S_2 are nested in n common loops, then it must be true that $\mathbf{i}_n' = \mathbf{j}_n'$ because $\mathbf{i}_n = \mathbf{j}_n$ and the transformation must be one-to-one. Therefore, $\mathbf{d}\,(\mathbf{i}', \mathbf{j}') = \mathbf{0}$, and there must exist a loop-independent dependence from S_2 to S_1, but not in the reverse direction. However, this violates the original dependence from S_1 to S_2—hence the transformation could not have been valid, and S_2 must follow S_1 in the transformed code.

WhileTheorem 2.4 initially appears to limit the possible transformations in the presence of loop-independent dependences,Theorem 2.5 shows that this is not so.

> **Theorem 2.5.** If S_2 has a loop-independent dependence on S_1, any reordering transformation that preserves their original order preserves that dependence.

Proof: By definition, S_2 and S_1 reference a location M on iteration vectors **i**, **j** such that **d** (**i**, **j**) = **0**, and S_2 follows S_1. A reordering transformation that maps **i** into **i′** and **j** into **j′** must have **d** (**i′**, **j′**) = **0**. If S_2 follows S_1, then the criteria for a loop-independent dependence are still met.

Given the properties of loop-independent dependences exhibited in Theorem 2.5, the natural extension of level notation utilized in loop carried dependence is to denote loop-independent dependence by an infinite level (that is, S_2 depends on S_1 with a loop independent dependence is denoted $S_1\ \delta_\infty S_2$). That level indicates that no ordering of loops can preserve the dependence.

Thus, any reordering transformation that preserves statement order preserves all loop-independent dependences. Theorem 2.3 and Theorem 2.5 illustrate clearly how loop-carried and loop-independent dependences complement each other. A loop-carried dependence is satisfied so long as certain loops are iterated in the correct order, regardless of the statement order within a specific iteration. A loop-independent dependence is satisfied so long as the statement order is maintained, regardless of the order in which the loops are iterated.

Loop-independent and loop-carried dependence completely partition all possible data dependences. To see this, it is only necessary to note that the existence of a dependence $S_1\ \delta\ S_2$ requires that S_1 be executed before S_2. This can happen in only two instances

1. when the difference vector for the dependence is less than **0**, or
2. when the difference vector equals **0** and S_1 occurs before S_2 textually.

These are precisely the criteria for loop-carried and loop-independent dependences, respectively. If neither of these is the case and S_1 and S_2 reference a common memory element, then S_2 is executed before S_1, and the dependence is actually $S_2\ \delta\ S_1$.

2.3 Application to Parallelization and Vectorization

2.3.1 Parallelization

Loop-carried dependences are precisely the concept necessary to determine whether or not the iterations of a loop may be safely executed in parallel without explicit synchronization.

> **Theorem 2.6. Fundamental Theorem of Loop Parallelization.** The iterations of a given sequential loop may be executed in parallel if the loop carries no dependence.

Proof. Once again, this follows directly from Bernstein's conditions and the observation that two different iterations of a single instantiation of a loop at level k will have iteration vectors that are equal in the positions corresponding to all the loops at levels 1 through k-1, and differ in the kth position. Hence, any dependence with source and sink in these two iterations must be carried by the level-k loop. Thus, by Bernstein's conditions, the parallelization is legal if the loop carries no dependence. QED.

2.3.2 Vectorization

Recall from Section 1.2.1 on page 19 that the task of vectorization is to determine whether statements in an inner loop can be vectorized by directly rewriting them in `FORTRAN 90`. Theorem 2.6 from the previous section tells us that any single-statement loop that carries no dependence can be directly vectorized, because that loop can be run in parallel. Thus the following loop.

```
DO I=1,N
   X(I) = X(I) + C
ENDDO
```

can be safely rewritten as:

```
X(1:N) = X(1:N) + C
```

On the other hand, in a slight permutation,

```
DO I=1,N
   X(I+1) = X(I) + C
ENDDO
```

which carries a dependence, the transformation to the statement

```
      X(2:N+1) = X(1:N) + C
```

would be incorrect since, on each iteration, the sequential version uses a value of `X` that is computed on the previous iteration, while the `FORTRAN 90` statement uses only old values of `X`.

Given the Fundamental Theorem of Loop Parallelization (Theorem 2.6), a natural question to ask is: "Can any statements in loops that carry dependences be directly vectorized?" This question is motivated by the following example:

```
    D0 I = 1, N
S1     A(I+1) = B(I) + C
S2     D(I) = A(I) + E
    ENDDO
```

This loop carries a dependence ($S_1 \, \delta \, S_2$) because it stores into `A` on one iteration and loads from `A` on the next. However, the direct transliteration of the loop into `FORTRAN 90` has the same meaning as the sequential loop.

```
S1  A(2:N+1) = B(1:N) + C
S2  D(1:N) = A(1:N) + E
```

The parallel version of the second statement uses elements `A(2:N)` defined by the previous loop, as it does in the sequential loop. This apparent contradiction can be explained by the fact that the process of transliterating actually slipped in an extra transformation, called *loop distribution*. The effect is as if the loop were first transformed into two different loops:

```
    D0 I = 1, N
S1     A(I+1) = B(I) + C
    ENDDO
    DO I = 1, N
S2     D(I) = A(I) + E
    ENDDO
```

each of which can be directly vectorized. At first, this case may lead one to believe that forward loop carried dependences do not inhibit vectorization, but that is not the complete truth, as the following example shows:

```
    DO I = 1, N
S2     D(I) = A(I) + E
S1     A(I+1) = B(I) + C
```

```
ENDDO
```

This example has a backward carried dependence. Obviously, this loop can be vectorized if the order of the statements are interchanged, because it then becomes identical to the previous case. The interchange is legal, since there are no loop-independent dependences between the statements.

The statement of this theorem is not quite correct; I'm wondering if we want to put a weak version here and strengthen it up (correctly) later. Looking at the algorithm, it corresponds to a weaker stint, so I've changed it.

Original stmt of theorem "if the statement is not included in any cycle consisting of loop-independent dependences and dependence carried by the loop.

The general conditions under which statements in a loop can be vectorized are given by the following theorem:

> **Theorem 2.7. Fundamental Theorem of Loop Vectorization.** A statement in a loop can be vectorized by directly rewriting in `FORTRAN 90` if the statement is not included in any cycle in the dependence graph.

Proof. If: Assume that the statement is not included in any cycle. Then the algorithm given in Figure 2.1 will correctly vectorize the statement.

Simple vectorization algorithm.

procedure *vectorize (L, D)*

/* L is the loop under examination for vectorization. */
/* D is the dependence graph for statements in L. */

find the set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D restricted to L (use Tarjan's algorithm[1]);

construct L_π from L by reducing each S_i to a single node and compute D_π, the dependence graph naturally induced on L_π by D;

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

for $i = 1$ **to** m **do**

if π_i is a dependence cycle **then**

generate a `DO`-loop around the statements in π_i

else

1. The algorithm referred to here is one for computing strongly connected regions in a graph. See \cite Tarjan.

```
            directly rewrite the single-statement πi in FORTRAN 90
        fi
    od
end
```

Recalling earlier discussions, the key semantic difference between a statement in a FORTRAN 77 DO−loop and its FORTRAN 90 vector analog is that the vector version must fetch all inputs before storing any outputs, while the DO−loop may intermix loads and stores. Thus, if a statement contained in a loop has all its inputs available at the beginning of the loop, that statement can be correctly rewritten in FORTRAN 90. The algorithm in Figure 2.1 groups the statements in the same loop into a totally ordered sequence of statement groups in which each statement is either part of a cycle in a dependence graph (also called a *recurrence*) or a single statement that is not part of a cycle. This reordering is equivalent to distributing loops around cycles and vectorizing statements that are not part of cycles. The distribution is correct, because there can be no backward edges from later cycles to earlier ones (topological ordering guarantees that), so that all necessary inputs will be available at the start of each distributed loop. The vectorization is legal because all the input values needed by a group from outside the group are available before the group begins executing, given that the ordering is consistent with dependence. As a result, any vectorization performed by the algorithm is correct. QED.

The previous proof goes only in the *if* direction, because in fact a weaker condition can be shown to hold in the *only if* direction, as the next section will show.

2.3.3 An Advanced Vectorization Algorithm

Theorem 2.7 states that any statement that does not depend upon itself, either directly or indirectly through a chain of dependences, can be vectorized. Thus, this theorem defines the cases when vectorization is safe.

However, in a nest of loops it is possible to construct a better algorithm based on the concept of dependence level and Theorem 2.3. Since a level-k dependence can be satisfied by running all loops outside of and including the loop at level k sequentially, it is often possible to eliminate a recurrence by running *some* loops sequentially. Theorem 2.3 suggests a recursive approach to the problem of multidimensional vectorization—first attempt to generate vector code at the outermost loop level. If dependences prevent that, then run the outer loop sequentially (thereby satisfying the dependences carried by that loop), and try again one level deeper, ignoring dependences carried by the outer loop. This approach is more rigorously defined in the procedure *codegen* (see Figure 2.2) [\citeK-enn^80, AllK^82].

FIGURE 2.2 Multi-level vector code generation algorithm.

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 R_π from R by reducing each S_i to a single node and compute D_π, the dependence graph naturally induced on R_π by D;

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

for i = 1 **to** m **do**

if π_i is cyclic **then**

generate a level-k `DO` statement;

let D_i be the dependence graph consisting of all dependence edges in D that are at level k+1 or greater and are internal to π_i;

codegen (π_i, k+1, D_i);

generate the level-k `CONTINUE` statement

else

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

fi

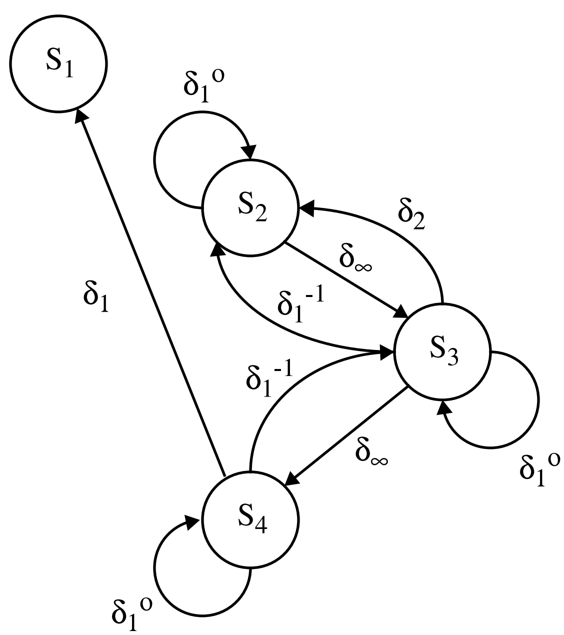
od

end

Codegen is called initially on a whole program at level 1 (the outermost level). The first step is to partition the program into *piblocks*, where a piblock is a strongly connected region as defined by Tarjan's algorithm \cite [Tarj^72,AhHU^74]. The definition of a strongly-connected region permits both cyclic and acyclic piblocks; however, any acyclic blocks are single statements that do not depend upon themselves. Next, the strongly connected regions are topologically sorted according to the dependence relation \cite [Knut^73]. Finally each region is examined in order. If the region is acyclic (thus necessarily consisting of one statement), then a parallel form of the statement is generated in the remaining dimensions. If the region is cyclic, the level 1 `DO`-loop is generated for that region, the level 1 dependences are deleted, and the process is repeated at level 2.

There is a missing level one dependence from S3 to S4 here. It's not important, because the loop independent dependence subsumes it, but it need to be mentioned.

Example dependence graph.



To illustrate the power of *codegen*, consider its application to the following program fragment.

```
    DO I = 1, 100
S1     X(I) = Y(I) + 10
       DO J = 1, 100
S2        B(J) = A(J,N)
```

```
            DO K = 1, 100
S3             A(J+1,K) = B(J) + C(J,K)
            ENDDO
S4          Y(I+J) = A(J+1, N)
         ENDDO
      ENDDO
```

The dependence graph for this program is given in Figure 2.3. The program contains the following dependences:

- There is both a level one and a loop-independent true dependence from S_2 to S_3 by virtue of the use of `B(J)` in both statements. The level one dependence is not as intuitive as the loop-independent one, but if the `I`-loop were to be run in parallel, later iterations of that loop could have S_2 store on top of a value of `B(J)` before S_3 fetched it.
- There is a loop-independent true dependence from S_3 to S_4 by virtue of the definition of `A(J+1,K)` in S_3 and the use of `A(J+1,N)` in statement S_4. Dependence is presumed to exist because, in the absence of other information about the value of `N`, it must be assumed that `N` can lie in the range of `K` (between 1 and 100). The conservative assumption is to assume that a dependence exists, unless it can be proven otherwise.
- There is a true dependence carried by the `I`-loop (level 1) from S_4 to S_1 because S_4 can store into `Y(I+1)` on one iteration and S_1 reads from it on the next iteration of the `I`-loop.
- There is a true dependence carried by the `J`-loop (level 2) from S_3 to S_2 because S_3 stores into `A(J+1,K)` on one iteration of the `J`-loop and reads from the same location via its access to `A(J,N)` on the next iteration. Once again, it must be assumed that `N` is between 1 and 100.
- Statements S_2, S_3 and S_4 all have output dependences upon themselves because their left-hand sides have one fewer dimension than the number of loops containing them. Hence, the outer loop must cause stores into the same array element on different iterations.
- There is an antidependence from S_4 to S_3, carried by the `I`-loop, because S_4 uses `A(J+1,N)` on one iteration of the `I`-loop; on a subsequent iteration of that loop (when the value of `J` is the same and `K=N`), S_3 stores into the same location.
- There is an antidependence from S_3 to S_2, carried by the `I`-loop, because S_3 uses `B(J)` and S_2 later stores into it on the next iteration of the `I`-loop.

Codegen called at the outermost level will produce two piblocks: one cyclic piblock consisting of S_2, S_3 and S_4, and one acyclic piblock consisting of S_1. As a result, S_1 will be vectorized, but must follow the code for the multi-statement piblock, due to topological ordering constraints. Thus, the code produced at this level is:

```
DO I = 1, 100
   codegen({S2, S3, S4}, 2})
ENDDO
X(1:100) = Y(1:100) + 10
```

In order to effect the call to *codegen* at level 2, all level-1 dependences are stripped off. As a result, parallel code is generated for S_4, but S_2 and S_3 are still in a recurrence. The code produced so far:

```
DO I = 1, 100
   DO J = 1, 100
      codegen({S2, S3}, 3})
   ENDDO
   Y(I+1:I+100) = A(2:101,N)
ENDDO
X(1:100) = Y(1:100) + 10
```

The final call to *codegen* requires that level-2 dependence be removed. After that removal, both remaining statements can be executed as vector operations. S_2 has no dimensions for the vector execution, so it will produces a simple scalar statement. The final code is given below:

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

Even though *codegen* is a very simple, elegant algorithm, it is able to effect powerful changes in code. In the example above, all available vectorization has been exposed, requiring a fairly dramatic change in statement order, by a straightforward application of the algorithm.

The proof of correctness of procedure *codegen* follows quite easily from the theorems in this chapter. If a region is cyclic, then Theorem 2.3 guarantees that any level-k dependences will be satisfied by running the k

loop (and by the nature of the algorithm, loops 1 through k-1) sequentially. Thus, the statements within the **then**-clause only ignore dependences that have been satisfied by the sequential loops. Once a level has been reached where all recurrences have been eliminated by sequentially executing DO loops, Theorem 2.7 guarantees that the remaining loops may be correctly run in parallel. Note that such a level *must* be reached (guaranteeing termination), since loop-independent dependences are inherently acyclic and all loop-carried dependences will eventually be satisfied, even if only by running all loops sequentially (thereby generating code that is parallel in 0 dimensions). Finally, the topological sort guarantees that loop-independent dependences will be preserved by virtue of Theorem 2.4 and Theorem 2.5. Although it has not been explicitly proved, it is also true that loop-carried dependences outside of a recurrence can be preserved by topological sorting without requiring that the statements be contained in common loops.

The first requirement for applying *codegen* to a loop nest is to have a dependence graph for that nest. The next chapter details tests for constructing that graph.

2.4 Summary

Because arrays can refer to more than one entity only inside of loops, the theory of dependence for arrays is closely related to loop iterations. In fact, the presence of loops give rise to a new type of dependence not normally encountered in scalar dependence—*loop-carried dependence*. Such dependences arise because of the iteration of specific loops. By capturing this information in the *level* of such a dependence, a broad range of reordering transformations become possible. In addition to loop-carried dependences, there are also *loop-independent* dependences, which are the type normally encountered in data-flow analysis.

One specific reordering transformation based on dependence is *vectorization*. The semantics of vector statements differ from the semantics of sequential statements in a manner very closely related to dependence. As a result, dependence graphs are extremely useful when translating sequential code to vector form. Figure 2.2 provides an efficient method for vectorization, once a dependence graph has been constructed. Methods for constructing the dependence graph are discussed in the next chapter.

2.5 Historical Notes

The earliest expositions of data dependence and its application to vectorization and program transformations wereby Lamport \cite{Lamport} and Kuck and Muraoka \cite{KMC:72,Muraoka:Dissertation}. Lamport developed the concepts of iteration spaces, difference vectors, a notion similar to direction vectors, and developed an algorithm applying these concepts to vectorization. He also had notions similar to true dependence, antidependence, and output dependence, although Kuck \cite{Kuck Book} was the first to precisely characterize and name these. Kuck and Muraoka created difference vectors and applied them to various forms of parallelism. Wolfe \cite {Wolfe Dissertation} developed the concept of direction vectors presented in this chapter, and applied them to a number of reordering transformations including loop interchange and vectorization.

The earliest algorithm for vectorization was one produced by Lamport \cite {Coordinate Method}, based on a notion of "tree consistency" and data dependence. Kuck and others developed a cleaner algorithm, based on dependence and self-dependence, that utilized Warshaw's algorithm to compute the transitive closure of the dependence relationship. Kennedy was the first to note the importance of cycles and the application of Tarjan's algorithm to this problem; the two algorithms presented in this book are due to him \cite {Allen and Kennedy TOPLAS paper}.

Allen was the first to precisely characterize reodering transformations as the range of transformations to which dependence could be applied. Many researchers have developed specific transformations based on dependence, including (list all from POPL Paper here) by Kuck et al \cite {Kuck POPL paper}, loop interchange \cite {Wolfe dissertation, Allen Kennedy 84 paper}, loop skewing \cite {Lamport, Wolfe Supercomputing paper}, parallelization \cite {Callahan dissertation, Munich POPL paper}, loop fusion \cite {Abu-Sufah dissertation, Warren POPL paper}.

The characterization of loop-carried dependences and loop-independent dependences was first noted by Allen and Kennedy \cite {Allen Disseration, Allen and Kennedy TOPLAS paper}. The properties of each classification, as well as the refined vectorization algorithm based on the level of dependences, was also developed by Allen and Kennedy.