Dependence: Theory and Practice

Allen and Kennedy, Chapter 2

Dependence: Theory and Practice

What shall we cover in this chapter?

- **Introduction to Dependences**
- **Loop-carried and Loop-independent Dependences**
- **Simple Dependence Testing**
- **Parallelization and Vectorization**

The Big Picture

What are our goals?

- Simple Goal: Make execution time as small as possible

Which leads to:

- Achieve execution of many (all, in the best case) instructions in parallel
- Find independent instructions

Dependences

- We will concentrate on data dependences
- Chapter 7 deals with control dependences

- Simple example of data dependence:

S_1 `PI = 3.14`

S_2 `R = 5.0`

S_3 `AREA = PI * R ** 2`

- Statement S_3 cannot be moved before either S_1 or S_2 without compromising correct results

Dependences

- **Formally:**

  There is a data dependence from statement S_1 to statement S_2 (S_2 depends on S_1) if:

  1. Both statements access the same memory location and at least one of them stores onto it, and
  2. There is a feasible run-time execution path from S_1 to S_2

Load Store Classification

- **Quick review of dependences classified in terms of load-store order:**
  1. **True dependences (RAW hazard)**
     - S_2 depends on S_1 is denoted by $S_1 \ \delta \ S_2$

  2. **Antidependence (WAR hazard)**
     - S_2 depends on S_1 is denoted by $S_1 \ \delta^{-1} \ S_2$

  3. **Output dependence (WAW hazard)**
     - S_2 depends on S_1 is denoted by $S_1 \ \delta^0 \ S_2$

Dependence in Loops

- Let us look at two different loops:

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

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

- In both cases, statement S_1 depends on itself
- However, there is a significant difference
- We need a formalism to describe and distinguish such dependences

Iteration Numbers

- The iteration number of a loop is equal to the value of the loop index

- Definition:
  - For an arbitrary loop in which the loop index I runs from L to U in steps of S, the iteration number *i* of a specific iteration is equal to the index value I on that iteration

Example:

```
      DO I = 0, 10, 2
S1        <some statement>
      ENDDO
```

Iteration Vectors

What do we do for nested loops?

- Need to consider the nesting level of a loop
- Nesting level of a loop is equal to one more than the number of loops that enclose it.
- Given a nest of n loops, the iteration vector *i* of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.
- Thus, the iteration vector is: $\{i_1, i_2, \ldots, i_n\}$
  where i_k, $1 \le k \le m$ represents the iteration number for the loop at nesting level k

Iteration Vectors

Example:

```
DO I = 1, 2
   DO J = 1, 2
S1       <some statement>
   ENDDO
ENDDO
```

- The iteration vector $S_1[(2, 1)]$ denotes the instance of S_1 executed during the 2nd iteration of the I loop and the 1st iteration of the J loop

Ordering of Iteration Vectors

- **Iteration Space: The set of all possible iteration vectors for a statement**

Example:

```
DO I = 1, 2
   DO J = 1, 2
S1       <some statement>
   ENDDO
ENDDO
```

- **The iteration space for S_1 is { (1,1), (1,2), (2,1), (2,2) }**

Ordering of Iteration Vectors

- Useful to define an ordering for iteration vectors
- Define an intuitive, lexicographic order
- Iteration i precedes iteration j, denoted $i < j$, iff:
  1. $i[1:n-1] < j[1:n-1]$, or
  2. $i[1:n-1] = j[1:n-1]$ and $i_n < j_n$

Formal Definition of Loop Dependence

- **Theorem 2.1 Loop Dependence:**
  There exists a dependence from statements S_1 to statement S_2 in a common nest of loops if and only if there exist two iteration vectors *i* and *j* for the nest, such that
  (1) $i < j$ or $i = j$ and there is a path from S_1 to S_2 in the body of the loop,
  (2) statement S_1 accesses memory location *M* on iteration *i* and statement S_2 accesses location *M* on iteration *j*, and
  (3) one of these accesses is a write.

- Follows from the definition of dependence

Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program

- But, what is the "meaning" of a program?

For our purposes:

- Two computations are equivalent if, on the same inputs:
  - They produce the same outputs in the same order

Reordering Transformations

- **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**

Properties of Reordering Transformations

- A reordering transformation does not eliminate dependences
- However, it can change the ordering of the dependence which will lead to incorrect behavior
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

Fundamental Theorem of Dependence

- Fundamental Theorem of Dependence:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program

- Proof by contradiction. Theorem 2.2 in the book.

Fundamental Theorem of Dependence

- A transformation is said to be *valid* for the program to which it applies if it preserves all dependences in the program.

Distance and Direction Vectors

- **Consider a dependence in a loop nest of n loops**
  - Statement S_1 on iteration i is the source of the dependence
  - Statement S_2 on iteration j is the sink of the dependence

- **The distance vector is a vector of length n d(i,j) such that: $d(i,j)_k = j_k - i_k$**

- We shall normalize distance vectors for loops in which the index step size is not equal to 1.

Direction Vectors

- **Definition 2.10 in the book:**

  Suppose that there is a dependence from statement S_1 on iteration *i* of a loop nest of *n* loops and statement S_2 on iteration *j*, then the *dependence direction vector* is $D(i,j)$ is defined as a vector of length *n* such that

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

Direction Vectors

Example:

```
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
```

- S_1 has a true dependence on itself.
- Distance Vector: (1, 0, -1)
- Direction Vector: (<, =, >)

Direction Vectors

- A dependence cannot exist if it has a direction vector whose leftmost non "=" component is not "<" as this would imply that the sink of the dependence occurs before the source.

Direction Vector Transformation

- Theorem 2.3. Direction Vector Transformation. 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 valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non- “=” component that is “>”.

- Follows from Fundamental Theorem of Dependence:
  - All dependences exist
  - None of the dependences have been reversed

Loop-carried and Loop-independent Dependences

- If in a loop statement S_2 depends on S_1, then there are two possible ways of this dependence occurring:

1. S_1 and S_2 execute on different iterations
   —This is called a loop-carried dependence.

2. S_1 and S_2 execute on the same iteration
   —This is called a loop-independent dependence.

Loop-carried dependence

- Definition 2.11
- 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).

Example:

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

Loop-carried dependence

- Level of a loop-carried dependence is the index of the leftmost non-“=” of D(i,j) for the dependence.

For instance:

```
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
```

- Direction vector for S1 is (=, =, <)
- Level of the dependence is 3
- A level-k dependence between S_1 and S_2 is denoted by $S_1\ \delta_k\ S_2$

Loop-carried Transformations

- Theorem 2.4 Any reordering transformation that does not alter the relative order of any loops in the nest and preserves the iteration order of the level-*k* loop preserves all level-*k* dependences.

- Proof:
  - D(i, j) has a "<" in the k^{th} position and "=" in positions 1 through k-1

  ⇒ Source and sink of dependence are in the same iteration of loops 1 through k-1

  ⇒ Cannot change the sense of the dependence by a reordering of iterations of those loops

- As a result of the theorem, powerful transformations can be applied

Loop-carried Transformations

Example:

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

can be transformed to:

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

Loop-independent dependences

- **Definition 2.14.** Statement S_2 has a *loop-independent dependence* on statement S_1 if and only if there exist two iteration vectors *i* and *j* such that:
  1) Statement S_1 refers to memory location *M* on iteration *i*, S_2 refers to *M* on iteration *j*, and $i = j$.
  2) There is a control flow path from S_1 to S_2 within the iteration.

Example:

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

Loop-independent dependences

More complicated example:

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

- No common loop is necessary. For instance:

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

Loop-independent dependences

- **Theorem 2.5.** If there is a loop-independent dependence from S_1 to S_2, any reordering transformation that does not move statement instances between iterations and preserves the relative order of S_1 and S_2 in the loop body preserves that dependence.

- S_2 depends on S_1 with a loop independent dependence is denoted by $S_1 \; \delta_\infty \, S_2$
- Note that the direction vector will have entries that are all "=" for loop independent dependences

Simple Dependence Testing

- **Theorem 2.7:** Let a and b be iteration vectors within the iteration space of the following loop nest:

```
DO i_1 = L_1, U_1, S_1
  DO i_2 = L_2, U_2, S_2
    ...
      DO i_n = L_n, U_n, S_n
S_1          A(f_1(i_1,...,i_n),...,f_m(i_1,...,i_n)) = ...
S_2          ... = A(g_1(i_1,...,i_n),...,g_m(i_1,...,i_n))
      ENDDO
     ...
    ENDDO
ENDDO
```

Simple Dependence Testing

```
DO i1 = L1, U1, S1
    DO i2 = L2, U2, S2
      ...
        DO in = Ln, Un, Sn
    S1         A(f1(i1,...,in),...,fm(i1,...,in)) = ...
    S2         ... = A(g1(i1,...,in),...,gm(i1,...,in))
        ENDDO
      ...
    ENDDO
ENDDO
```

- A dependence exists from S_1 to S_2 if and only if there exist values of α and β such that (1) α is lexicographically less than or equal to β and (2) the following system of *dependence equations* is satisfied:

  $f_i(\alpha) = g_i(\beta)$ for all i, $1 \le i \le m$

- Direct application of Loop Dependence Theorem

Simple Dependence Testing: Delta Notation

- Notation represents index values at the source and sink

  Example:

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

- Iteration at source denoted by: I_0
- Iteration at sink denoted by: $I_0 + \Delta I$
- Forming an equality gets us: $I_0 + 1 = I_0 + \Delta I$
- Solving this gives us: $\Delta I = 1$

  $\Rightarrow$ Carried dependence with distance vector (1) and direction vector (<)

Simple Dependence Testing: Delta Notation

Example:

```
DO I = 1, 100
    DO J = 1, 100
        DO K = 1, 100
            A(I+1,J,K) = A(I,J,K+1) + B
        ENDDO
    ENDDO
ENDDO
```

- $I_0 + 1 = I_0 + \Delta I$; $J_0 = J_0 + \Delta J$; $K_0 = K_0 + \Delta K + 1$
- Solutions: $\Delta I = 1$; $\Delta J = 0$; $\Delta K = -1$
- Corresponding direction vector: (<, =, >)

Simple Dependence Testing: Delta Notation

- If a loop index does not appear, its distance is unconstrained and its direction is "*"

  Example:

```
DO I = 1, 100
    DO J = 1, 100
        A(I+1) = A(I) + B(J)
    ENDDO
ENDDO
```

- The direction vector for the dependence is (<, *)

Simple Dependence Testing: Delta Notation

- * denotes union of all 3 directions

Example:

```
DO J = 1, 100
    DO I = 1, 100
            A(I+1) = A(I) + B(J)
    ENDDO
ENDDO
```

- (*, <) denotes { (<, <), (=, <), (>, <) }
- Note: (>, <) denotes a level 1 antidependence with direction vector (<, >)

Parallelization and Vectorization

- Theorem 2.8. It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

- Want to convert loops like:

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

- to `X(1:N) = X(1:N) + C` (Fortran 77 to Fortran 90)

- However:

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

is not equivalent to `X(2:N+1) = X(1:N) + C`

Loop Distribution

- Can statements in loops which carry dependences be vectorized?

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

- Dependence: $S_1\ \delta_1\ S_2$ can be converted to:

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

Loop Distribution

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

- transformed to:

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

- leads to:

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

Loop Distribution

- Loop distribution fails if there is a cycle of dependences

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

$S_1 \; \delta_1 \; S_2$ and $S_2 \; \delta_1 \; S_1$

- What about:

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