15-745 Lecture 6

Data Dependence in Loops

Based on slides from Allen&Kennedy

Common loop optimizations

- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to to improve scheduling of the loop body
- Software pipelining
  - To improve scheduling of the loop body
- Loop permutation
  - to improve cache memory performance

Requires understanding data dependencies

Why Dependence Analysis

- Goal is to find best schedule:
  - Improve memory locality
  - Increase parallelism
  - Decrease scheduling stalls
- Before we schedule we need to know possible legal schedules and impact of schedule on performance

Example to improve locality

```
for i=0 to N
  for j=0 to M
    A[j] = f(A[j]);
```

Is there a better schedule?

<u>Unroll to see deps</u>

```
A[0] = f(A[0])
A[1] = f(A[1])
A[2] = f(A[2])
...
A[N] = f(A[N])
A[0] = f(A[0])
...
```

Iteration space

i

A[0] ← A[0]	A[1] ← A[1]	A[2] ← A[2]	A[3] ← A[3]
A[0] ← A[0]	A[1] ← A[1]	A[2] ← A[2]	A[3] ← A[3]
A[0] ← A[0]	A[1] ← A[1]	A[2] ← A[2]	A[3] ← A[3]
A[0] ← A[0]	A[1] ← A[1]	A[2] ← A[2]	A[3] ← A[3]

j

Example to improve locality

```
for i=0 to N
  for j=0 to M
    A[j] = f(A[j]);
```

Is there a better schedule?

Unroll to see deps

A[0] = f(A[0])
A[1] = f(A[1])
A[2] = f(A[2])
...
A[N] = f(A[N])
A[0] = f(A[0])
...

Iteration space

i ↑

A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]
A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]
A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]
A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]

j ⟶

```
for j=0 to M
  for i=0 to N
    A[j] = f(A[j]);
```


Transformed iteration space

Old Iteration space

```
for i=0 to N
  for j=0 to M
    A[j] = f(A[j]);
```

i ↑

A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]
A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]
A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]
A[0]←A[0]	A[1]←A[1]	A[2]←A[2]	A[3]←A[3]

j ⟶

New Iteration space

```
for j=0 to M
  for i=0 to N
    A[j] = f(A[j]);
```

j ↑

A[3]←A[3]	A[3]←A[3]	A[3]←A[3]	A[3]←A[3]
A[2]←A[2]	A[2]←A[2]	A[2]←A[2]	A[2]←A[2]
A[1]←A[1]	A[1]←A[1]	A[1]←A[1]	A[1]←A[1]
A[0]←A[0]	A[0]←A[0]	A[0]←A[0]	A[0]←A[0]

i ⟶

What about …

```
for i=0 to N
  for j=0 to M
    A[j] = f(A[j]);
    B[i] = f(B[i]);
```

Is there a better schedule?

Unroll to see deps

A[0] = f(A[0])
B[0] = f(B[0])
A[1] = f(A[1])
B[0] = f(B[0]])
...
A[N] = f(A[N])
B[0] = f(B[0])
A[0] = f(A[0])
B[1] = f(B[1])
...

Iteration space

i ↑

A[0]←A[0] B[3]←B[3]	A[1]←A[1] B[3]←B[3]	A[2]←A[2] B[3]←B[3]	A[3]←A[3] B[3]←B[3]
A[0]←A[0] B[2]←B[2]	A[1]←A[1] B[2]←B[2]	A[2]←A[2] B[2]←B[2]	A[3]←A[3] B[2]←B[2]
A[0]←A[0] B[1]←B[1]	A[1]←A[1] B[1]←B[1]	A[2]←A[2] B[1]←B[1]	A[3]←A[3] B[1]←B[1]
A[0]←A[0] B[0]← B[0]	A[1]←A[1] B[0]← B[0]	A[2]←A[2] B[0]← B[0]	A[3]←A[3] B[0]← B[0]

J ⟶

What about …

```
for i=0 to N
  for j=0 to M
    A[j] = f(A[j]);
    B[i] = f(B[i]);
```

Is there a better schedule?

Unroll to see deps

A[0] = f(A[0])
B[0] = f(B[0])
A[1] = f(A[1])
B[0] = f(B[0]])
...
A[N] = f(A[N])
B[0] = f(B[0])
A[0] = f(A[0])
B[1] = f(B[1])
...

Iteration space

i ↑

A[0]←A[0] B[3]←B[3]	A[1]←A[1] B[3]←B[3]	A[2]←A[2] B[3]←B[3]	A[3]←A[3] B[3]←B[3]
A[0]←A[0] B[2]←B[2]	A[1]←A[1] B[2]←B[2]	A[2]←A[2] B[2]←B[2]	A[3]←A[3] B[2]←B[2]
A[0]←A[0] B[1]←B[1]	A[1]←A[1] B[1]←B[1]	A[2]←A[2] B[1]←B[1]	A[3]←A[3] B[1]←B[1]
A[0]←A[0] B[0]← B[0]	A[1]←A[1] B[0]← B[0]	A[2]←A[2] B[0]← B[0]	A[3]←A[3] B[0]← B[0]

J ⟶

But, what if …

```
for i=0 to N
  for j=1 to M
    A[j] = f(A[j-1]);
```

Can we reschedule?

Iteration space

A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]

i

j

<u>Unroll to see deps</u>

A[1] = f(A[0])
A[2] = f(A[1])
A[3] = f(A[2])
…
A[N] = f(A[N-1])
A[1] = f(A[0])
A[2] = f(A[1])
A[3] = f(A[2])
…

But, what if …

```
for i=0 to N
  for j=1 to M
    A[j] = f(A[j-1]);
```

Can we reschedule?

Iteration space

A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]

i

j

But, what if …

```
for i=0 to N
  for j=1 to M
    A[j] = f(A[j-1]);
```

Can we reschedule?

Iteration space

A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]

i

j

			A[1]←A[0]	A[2]←A[1]	A[3]←A[2]
		A[] 1←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]
	A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]	
A[1]←A[0]	A[2]←A[1]	A[3]←A[2]	A[4]←A[3]		

So, how do we know when/how?

When should we transform a loop?
What transforms are legal?
How should we transform the loop.

Dependence information helps with all three questions.

In short,

- Determine all dependence information
- Use dependence information to analyze loop
- Guide transformations using dependence info

- Key is:
  Any transformation* that preserves every dependence in a program preserves the meaning of the program

Dependencies in Loops

- Loop independent data dependence occurs between accesses in the same loop iteration.
- Loop-carried data dependence occurs between accesses across different loop iterations.
- There is data dependence between
  access a at iteration i-k and
  access b at iteration i when:
  - a and b access the same memory location
  - There is a path from a to b
  - Either a or b is a write

Defining Dependencies

- Flow Dependence W → R δ^f } true
- Anti-Dependence R → W δ^a
- Output Dependence W → W δ^o } false

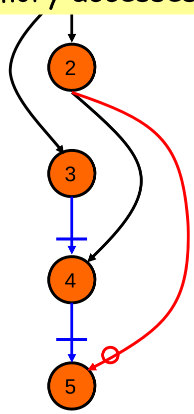
```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```


Example Dependencies

```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

These are scalar dependencies. The same idea holds for memory accesses.

source	type	target	due to
S1	δ^f	S2	a
S1	δ^f	S3	a
S2	δ^f	S4	b
S3	δ^a	S4	d
S4	δ^a	S5	b
S2	δ^o	S5	b



What can we do with this information?
What are anti- and flow- called "false" dependences?

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
     A[i] = B[i];
     B[i+1] = A[i];
}
```

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

δ^f loop carried

```
for (i=0; i<n; i++) {
        A[i] = B[i];
        B[i+1] = A[i];
}
```

δ^f loop independent

Data Dependence

- 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

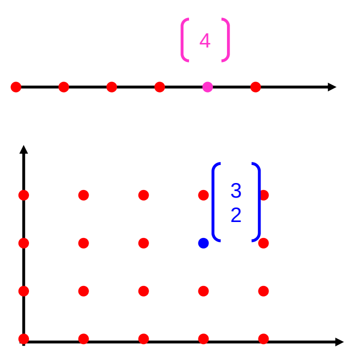
- We need to characterize the dependence information in terms of the loop iterations involved in the dependence, so we need a way to talk about iterations of a loop.
  - Iteration vector: a label for a loop iteration using the induction variables.
  - Iteration space: the set of all possible iteration vectors for a loop
  - Lexicographic order: The order of the iterations

Iteration Space

Every iteration generates a point in an n-dimensional space, where n is the depth of the loop nest.

```
for (i=0; i<n; i++) {
    •••
}
```

```
for (i=0; i<n; i++)
    for (j=0; j<4; j++) {
        •••
}
```



T. Mowry

Iteration Vectors

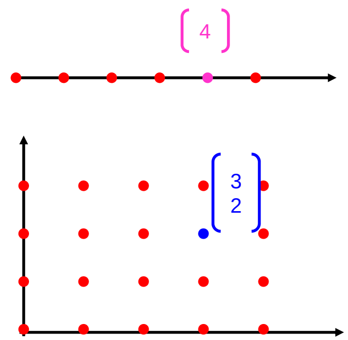
- Need to consider the nesting level of a loop
- Nesting level of a loop is equal to one more than the number of loops that enclose it.
- Given a nest of n loops, the iteration vector *i* of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each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

Iteration Space

Every iteration generates a point in an n-dimensional space, where n is the depth of the loop nest.

```
for (i=0; i<n; i++) {
    •••
}
```

```
for (i=0; i<n; i++)
    for (j=0; j<4; j++) {
        •••
}
```



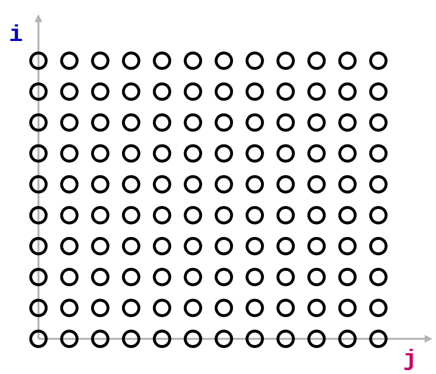
Ordering of Iteration Vectors

- Dan ordering for iteration vectors
- Use 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:k-1] = j[1:k-1] and $i_k < j_k$

$$\begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_k \\ \dots \\ i_n \end{pmatrix} < \begin{pmatrix} j_1 \\ j_2 \\ \dots \\ j_k \\ \dots \\ j_n \end{pmatrix}$$

Example Iteration Space

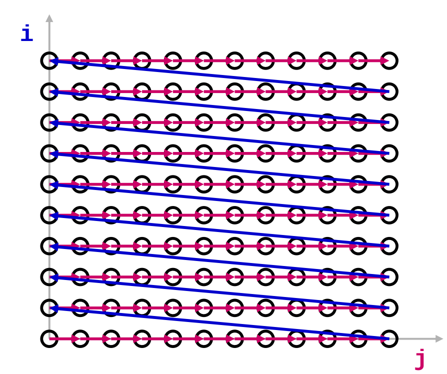
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- Note: iteration space is not data space

Formal Def of Loop Dependence

- There exists a dependence from statements S_1 to statement S_2 in a common nest of loops iff there exist two iteration vectors *i* and *j* for the nest, st.
  (1) (a) $i < j$ or
  (b) $i = j$ 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.
- 1a: Loop carried and 1b: Loop independent
- S1 is source of dependence, S2 is sink or target of dep

Dependence Distance

- Using iteration vectors and def of dependence we can determine the distance of a dependence:
- In n-deep loop nest if
  - S1 is source in iteration i
  - S2 is sink in iteration j
- Distance of dependence is represented with a distance vector: D
  - Vector of length n, where
  - $d_k = j_k - i_k$

Distance Vector

```
for (i=0; i<n; i++) {
     A[i] = B[i];
     B[i+1] = A[i];
}
```

```
A[0] = B[0];
B[1] = A[0];     } i=0
A[1] = B[1];
B[2] = A[1];     } i=1
A[2] = B[2];
B[3] = A[2];     } i=2
   ⋮
```

Distance vector is the difference between the target and source iterations.

$$\mathbf{d = I_t - I_s}$$

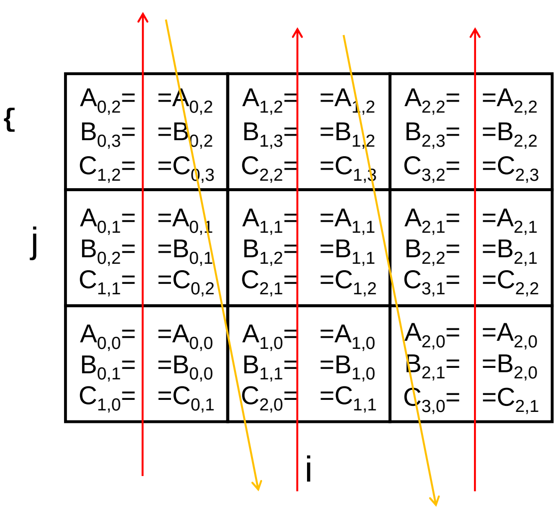
Exactly the distance of the dependence, i.e.,

$$\mathbf{I_s + d = I_t}$$

T. Mowry

Example of Distance Vectors

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++){
    A[i,j] =    ;
        = A[i,j];
    B[i,j+1] =    ;
        = B[i,j];
    C[i+1,j] =    ;
        = C[i,j+1] ;
  }
```



T. Mowry

Example of Distance Vectors

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++){
    A[i,j] =   ;
        = A[i,j];
    B[i,j+1] =   ;
        = B[i,j];
    C[i+1,j] =   ;
        = C[i,j+1] ;
  }
```

j

$A_{0,2}=\ \ =A_{0,2}$ $B_{0,3}=\ \ =B_{0,2}$ $C_{1,2}=\ \ =C_{0,3}$	$A_{1,2}=\ \ =A_{1,2}$ $B_{1,3}=\ \ =B_{1,2}$ $C_{2,2}=\ \ =C_{1,3}$	$A_{2,2}=\ \ =A_{2,2}$ $B_{2,3}=\ \ =B_{2,2}$ $C_{3,2}=\ \ =C_{2,3}$
$A_{0,1}=\ \ =A_{0,1}$ $B_{0,2}=\ \ =B_{0,1}$ $C_{1,1}=\ \ =C_{0,2}$	$A_{1,1}=\ \ =A_{1,1}$ $B_{1,2}=\ \ =B_{1,1}$ $C_{2,1}=\ \ =C_{1,2}$	$A_{2,1}=\ \ =A_{2,1}$ $B_{2,2}=\ \ =B_{2,1}$ $C_{3,1}=\ \ =C_{2,2}$
$A_{0,0}=\ \ =A_{0,0}$ $B_{0,1}=\ \ =B_{0,0}$ $C_{1,0}=\ \ =C_{0,1}$	$A_{1,0}=\ \ =A_{1,0}$ $B_{1,1}=\ \ =B_{1,0}$ $C_{2,0}=\ \ =C_{1,1}$	$A_{2,0}=\ \ =A_{2,0}$ $B_{2,1}=\ \ =B_{2,0}$ $C_{3,0}=\ \ =C_{2,1}$

i

A yields: $\begin{pmatrix}0\\0\end{pmatrix}$ B yields: $\begin{pmatrix}0\\1\end{pmatrix}$ C yields: $\begin{pmatrix}1\\-1\end{pmatrix}$

Direction Vectors

- Less precise than distance vectors, but often good enough
- In n-deep loop nest if
  - S1 is source in iteration i
  - S2 is sink in iteration j
- Distance vector: F
  - Vector of length n, where
  - $f_k = j_k - i_k$
- Direction vector also vector of length n, where
  - $d_k = \begin{cases} \text{"<" if } f_k > 0, \text{ or } j_k < i_k \\ \text{"=" if } f_k = 0, \text{ or } j_k = i_k \\ \text{">" if } f_k < 0, \text{ or } j_k > i_k \end{cases}$

Example of Direction Vectors

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++){
    A[i,j] =   ;
        = A[i,j];
    B[i,j+1] =   ;
        = B[i,j];
    C[i+1,j] =   ;
        = C[i,j+1] ;
  }
```

j

$A_{0,2}=\ \ =A_{0,2}$ $B_{0,3}=\ \ =B_{0,2}$ $C_{1,2}=\ \ =C_{0,3}$	$A_{1,2}=\ \ =A_{1,2}$ $B_{1,3}=\ \ =B_{1,2}$ $C_{2,2}=\ \ =C_{1,3}$	$A_{2,2}=\ \ =A_{2,2}$ $B_{2,3}=\ \ =B_{2,2}$ $C_{3,2}=\ \ =C_{2,3}$
$A_{0,1}=\ \ =A_{0,1}$ $B_{0,2}=\ \ =B_{0,1}$ $C_{1,1}=\ \ =C_{0,2}$	$A_{1,1}=\ \ =A_{1,1}$ $B_{1,2}=\ \ =B_{1,1}$ $C_{2,1}=\ \ =C_{1,2}$	$A_{2,1}=\ \ =A_{2,1}$ $B_{2,2}=\ \ =B_{2,1}$ $C_{3,1}=\ \ =C_{2,2}$
$A_{0,0}=\ \ =A_{0,0}$ $B_{0,1}=\ \ =B_{0,0}$ $C_{1,0}=\ \ =C_{0,1}$	$A_{1,0}=\ \ =A_{1,0}$ $B_{1,1}=\ \ =B_{1,0}$ $C_{2,0}=\ \ =C_{1,1}$	$A_{2,0}=\ \ =A_{2,0}$ $B_{2,1}=\ \ =B_{2,0}$ $C_{3,0}=\ \ =C_{2,1}$

i

A yields: $\begin{pmatrix}=\\=\end{pmatrix}$ B yields: $\begin{pmatrix}=\\<\end{pmatrix}$ C yields: $\begin{pmatrix}<\\>\end{pmatrix}$

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: (<, =, >)

Note on 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.
- Likewise, the first non-zero distance in a distance vector must be postive.

The Key

- Any reordering transformation that preserves every dependence in a program preserves the meaning of the program
- A reordering transformation may change order of execution but does not add or remove statements.

Finding Data Dependences

Main Theme

- Determining whether dependencies exist between two subscripted references to the same array in a loop nest
- Several tests to detect these dependencies

The General Problem

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

A dependence exists from S1 to S2 if:

- There exist α and β such that
  - $\alpha < \beta$ (control flow requirement)
  - $f_i(\alpha) = g_i(\beta)$ for all i, $1 \le i \le m$ (common access requirement)

Basics: Conservative Testing

- Consider only linear subscript expressions
- Finding integer solutions to system of linear Diophantine Equations is NP-Complete
- Most common approximation is Conservative Testing, i.e., See if you can assert

  "No dependence exists between two subscripted references of the same array"
- Never incorrect, may be less than optimal

Basics: Indices and Subscripts

Index: Index variable for some loop surrounding a pair of references

Subscript: A PAIR of subscript positions in a pair of array references

For Example:

```
A(I,j) = A(I,k) + C
```

<I,I> is the first subscript

<j,k> is the second subscript

Basics: Complexity

A subscript is said to be

- ZIV if it contains no index
  zero index variable
- SIV if it contains only one index
  single index variable
- MIV if it contains more than one index
  multiple index variable

For Example:

```
A(5,I+1,j) = A(1,I,k) + C
      First subscript is ZIV
      Second subscript is SIV
      Third subscript is MIV
```

Basics: Separability

- A subscript is separable if its indices do not occur in other subscripts
- If two different subscripts contain the same index they are coupled

For Example:

```
A(I+1,j) = A(k,j) + C
```

Both subscripts are separable

```
A(I,j,j) = A(I,j,k) + C
```

Second and third subscripts are coupled

Basics:Coupled Subscript Groups

- Why are they important?

  Coupling can cause imprecision in dependence testing

```
   DO I = 1, 100
S1   A(I+1,I) = B(I) + C
S2   D(I) = A(I,I) * E
   ENDDO
```

Dependence Testing: Overview

- Partition subscripts of a pair of array references into separable and coupled groups
- Classify each subscript as ZIV, SIV or MIV
  - Reason for classification is to reduce complexity of the tests.
- For each separable subscript apply single subscript test. Continue until prove independence.
- Deal with coupled groups
- If independent, done
- Otherwise, merge all direction vectors computed in the previous steps into a single set of direction vectors

Step 1: Subscript Partitioning

- Partitions the subscripts into separable and minimal coupled groups
- Notations

  // S is a set of m subscript pairs $S_1, S_2, \ldots S_m$ each enclosed in n loops with indexes $I_1, I_2, \ldots I_n$, which is to be partitioned into separable or minimal coupled groups.

  // P is an output variable, containing the set of partitions

  // n_p is the number of partitions

Subscript Partitioning Algorithm

```
procedure partition(S,P, n_p)
    n_p = m;
    for i  :=  1 to m do P_i = {S_i};
    for i  :=  1 to n do begin
        k := <none>
        for each remaining partition P_j do
            if there exists s ε P_j such that s contains I_i then
                if k = < none > then k = j;
                else begin P_k = P_k ∪ P_j; discard Pj; n_p = n_p – 1; end
    end
end partition
```

Step 2: Classify as ZIV/SIV/MIV

- Easy step
- Just count the number of different indices in a subscript

Step 3: Applying Single Subscript Tests

- ZIV Test
- SIV Test
  - Strong SIV Test
  - Weak SIV Test
    - Weak-zero SIV
    - Weak Crossing SIV
- SIV Tests in Complex Iteration Spaces

ZIV Test

```
 DO j = 1, 100
S      A(e1) = A(e2) + B(j)
 ENDDO
```

e1,e2 are constants or loop invariant symbols

If (e1-e2)!=0 No Dependence exists

Strong SIV Test

- Strong SIV subscripts are of the form

$$\langle ai + c_1, ai + c_2 \rangle$$

- For example the following are strong SIV subscripts

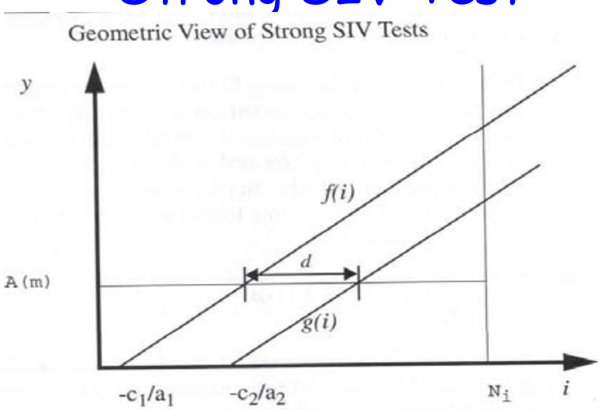
$$\langle i+1, i \rangle$$

$$\langle 4i+2, 4i+4 \rangle$$

Strong SIV Test Example

```
DO k = 1, 100
  DO j = 1, 100
S1      A(j+1,k) = ...
S2           ... = A(j,k) + 32
  ENDDO
ENDDO
```

Strong SIV Test



$$d = i' - i = \frac{c_1 - c_2}{a}$$

Dependence exists if $|d| \le U - L$

Weak SIV Tests

- Weak SIV subscripts are of the form

$$\langle a_1 i + c_1, a_2 i + c_2 \rangle$$

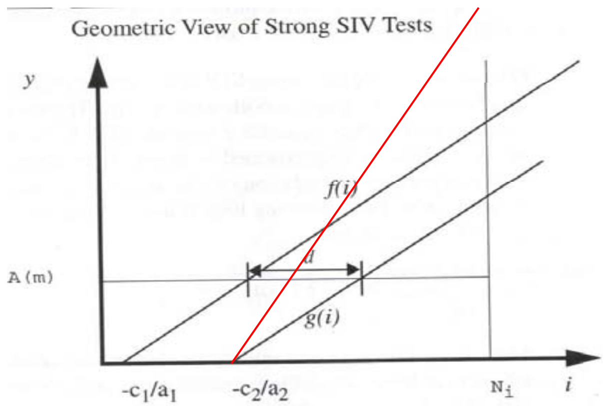
- For example the following are weak SIV subscripts

$$\langle i+1, 5 \rangle$$

$$\langle 2i+1, i+5 \rangle$$

$$\langle 2i+1, -2i \rangle$$

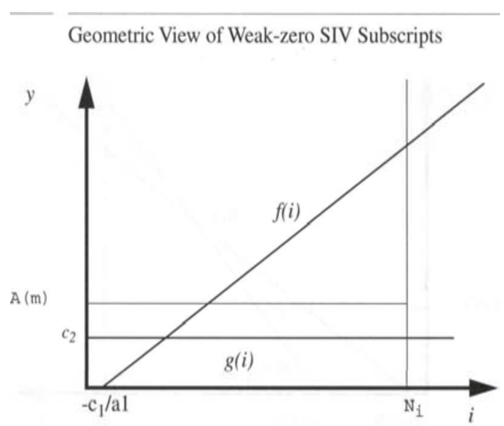
Geometric view of weak SIV



Weak-zero SIV Test

- Special case of Weak SIV where one of the coefficients of the index is zero
- The test consists merely of checking whether the solution is an integer and is within loop bounds $i = \frac{c_2 - c_1}{a_1}$

Weak-zero SIV Test



Weak-zero SIV & Loop Peeling

```
     DO i = 1, N
S1     Y(i, N) = Y(1, N) + Y(N, N)
     ENDDO
```

Can be loop peeled to...

```
     Y(1, N) = Y(1, N) + Y(N, N)
     DO i = 2, N-1
S1     Y(i, N) = Y(1, N) + Y(N, N)
     ENDDO
     Y(N, N) = Y(1, N) + Y(N, N)
```

Weak-crossing SIV Test

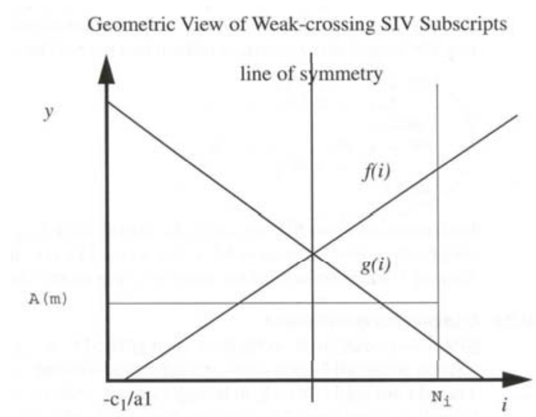
- Special case of Weak SIV where the coefficients of the index are equal in magnitude but opposite in sign
- The test consists merely of checking whether the solution index

  is 1. within loop bounds and is

  2. either an integer or has a non-integer part equal to 1/2

$$i = \frac{c_2 - c_1}{2a_1}$$

Weak-crossing SIV Test



Weak-crossing SIV & Loop Splitting

```
     DO i = 1, N
S1   A(i) = A(N-i+1) + C
     ENDDO
```

This loop can be split into…

```
DO i = 1,(N+1)/2
   A(i) = A(N-i+1) + C
ENDDO
DO i = (N+1)/2 + 1, N
   A(i) = A(N-i+1) + C
ENDDO
```

Complex Iteration Spaces

- Till now we have applied the tests only to rectangular iteration spaces
- These tests can also be extended to apply to triangular or trapezoidal loops
  - Triangular: One of the loop bounds is a function of at least one other loop index
  - Trapezoidal: Both the loop bounds are functions of at least one other loop index

Next Time...

- Complex iteration spaces
- MIV Tests
- Tests in Coupled groups
- Merging direction vectors