CS 610: Data Dependence Analysis

Swarnendu Biswas

Semester 2022-2023-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

How to Write Efficient and Scalable Programs?

Good choice of algorithms and data structures

- Determines number of operations executed

Code that the compiler and architecture can effectively optimize

- Determines number of instructions executed

Proportion of parallelizable and concurrent code

- Amdahl's law

Sensitive to the architecture platform

- Efficiency and characteristics of the platform
- For e.g., memory hierarchy, cache sizes

Role of a Good Compiler

Try and extract performance automatically

Optimize memory access latency

- Code restructuring optimizations
- Prefetching optimizations
- Data layout optimizations
- Code layout optimizations

Parallelism Challenges for a Compiler

- On single-core machines
  - Focus is on register allocation, instruction scheduling, reduce the cost of array accesses
- On parallel machines
  - Find parallelism in sequential code, find portions of work that can be executed in parallel
  - Principle strategy is data decomposition – good idea since this can scale

Can we parallelize the following loops?

```
do i = 1, 100
  A(i) = A(i) + 1
enddo
```

```
do i = 1, 100
  A(i) = A(i-1) + 1
enddo
```

- Focus is on loop parallelism because it can provide more benefits
  - Inter-statement or and intra-statement parallelism is limited

Parallelism and Data Dependence

- Compiler should apply transformations only when it is **safe** to do so

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.

- A reordering transformation that preserves every dependence preserves the meaning of the program
- Parallel loop iterations imply random interleaving of statements in the loop body

Data Dependences

```
S1   a = b + c
S2   d = a * 2
S3   a = c + 2
S4   e = d + c + 2
```

Execution constraints

- S2 must execute after S1
- S3 must execute after S2
- S3 must execute after S1
- S3 and S4 can execute in any order, and concurrently

There is a data dependence from S1 to S2 if and only if

- Both statements access the same memory location
- At least one of the accesses is a write
- There is a feasible execution path at run-time from S1 to S2

Types of Dependences

Flow (true) or RAW hazard

Denoted by $S_1 \delta S_2$

Anti or WAR hazard

Denoted by $S_1 \delta^{-1} S_2$

Output or WAW hazard

Denoted by $S_1 \delta^{o} S_2$

Input

```
S1   X = …
S2   … = X

S1   … = X
S2   X = …

S1   X = …
S2   X = …

S1   … = a/b
S2   … = b * c
```

Bernstein's Conditions

- Suppose there are two processes P_1 and P_2
- Let I_i be the set of all input variables for process P_i
- Let O_i be the set of all output variables for process P_i

- P_1 and P_2 can execute in parallel (denoted as $P_1 \,||\, P_2$) if and only if
  - $I_1 \cap I_2 = \Phi$
  - $I_2 \cap O_1 = \Phi$
  - $O_2 \cap O_1 = \Phi$

Bernstein's Conditions

- Suppose there are two processes P_1 an
- Let vari
- Let O_i be the set of all output variables for process P_i

- P_1 and P_2 can execute in parallel ... if and only
  - $O_2 \cap O_1 = \Phi$

Two processes can execute in parallel if they are flow-, anti-, and output-independent

Bernstein's Conditions

- Suppose there are two processes P_1 ar
- Let I
  varia
- Let O_i be the set of all output variables for process P_i

- P_1 and P_2 can execute in parallel
  only
  - $O_2 \cap O_1 = \Phi$

Two processes can execute in parallel if they are flow-, anti-, and output-independent

- If P_i || P_j, does that imply P_j || P_i?
- If P_i || P_j and P_j || P_k, does that imply P_i || P_k?

Find Parallelism in Loops – Is it Easy?

- Need to analyze array subscripts
- Need to check whether two array subscripts access the same memory location

```
   for i = 1 to N
S1    A[i+1] = A[i] + B[i]
   endfor
```

```
   for i = 1 to N
S1    A[i+2] = A[i] + B[i]
   endfor
```

- Statement S1 depends on itself in both examples, however, there is a significant difference
- Compilers need formalism to describe and distinguish dependences

Enumerate All Dependences in Loops

```
    for i = 1 to 50
S1    A[i] = B[i-1] + C[i]
S2    B[i] = A[i+2] + C[i]
    endfor
```

- Unrolling loops can help figure out dependences

```
S1(1)    A[1] = B[0] + C[1]
S2(1)    B[1] = A[3] + C[1]
S1(2)    A[2] = B[1] + C[2]
S2(2)    B[2] = A[4] + C[2]
S1(3)    A[3] = B[2] + C[3]
S2(3)    B[3] = A[5] + C[3]
```

- large loop bounds
- loop bounds may not be known at compile time

Normalized Iteration Number

- Parameterize the statement with the loop iteration number

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

     DO I = L, U, S
S1     ...
     ENDDO
```

For a loop where the loop index *I* runs from *L* to *U* in steps of *S*, the *normalized iteration number* of a specific iteration is $(I–L+1)/S$, where *I* is the value of the index on that iteration

Iteration Vector

Given a nest of n loops, the *iteration vector* ***i*** of an iteration of the innermost loop is a vector of integers containing the iteration numbers for each of the loops in order of nesting level.

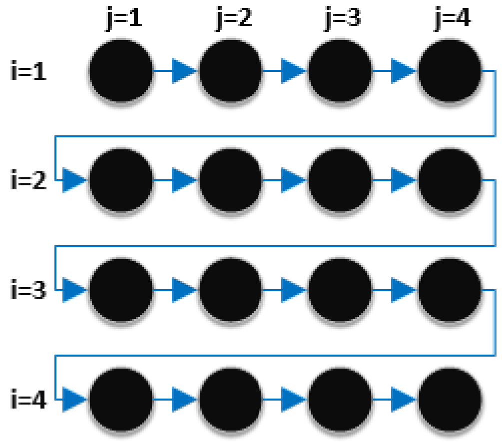
The iteration vector ***i*** is $\{i_1, i_2, ..., i_n\}$ where i_k, $1 \leq k \leq n$, represents the iteration number for the loop at nesting level k.

- A vector (d1, d2) is positive if (0,0) < (d1, d2), i.e., its first (leading) non-zero component is positive

Iteration Space Graphs

- Represent each dynamic instance of a loop as a point in the graph
- Draw arrows from one point to another to represent dependences

```
      for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
S1:       a[i][j] = a[i][j-1] * x;
```



Dimension of iteration space is the loop nest level, need not always be rectangular

```
for i = 1 to 5 do
  for j = i to 5 do
    A(i, j) = B(i, j) + C(j)
  endfor
endfor
```

Lexicographic Ordering of Iteration Vectors

- Assume i is a vector, i_k is the k^{th} element of the vector i, and i[1:k] is a k-vector consisting of the leftmost k elements of i

- Iteration i precedes iteration j, denoted by $i < j$, if and only if
  i. $i[1:n-1] < j[1:n-1]$, or
  ii. $i[1:n-1] = j[1:n-1]$ and $i_n < j_n$

Formal Definition of Loop Dependence

There exists a loop dependence from statement S1 to S2 in a loop nest if and only if there exist two iteration vectors i and j for the nest, such that

i. i < j or i = j and there is a path from S1 to S2 in the body of the loop,
ii. S1 accesses memory location *M* on iteration i and S2 accesses *M* on iteration j, and
iii. one of these accesses is a write.

Distance Vectors

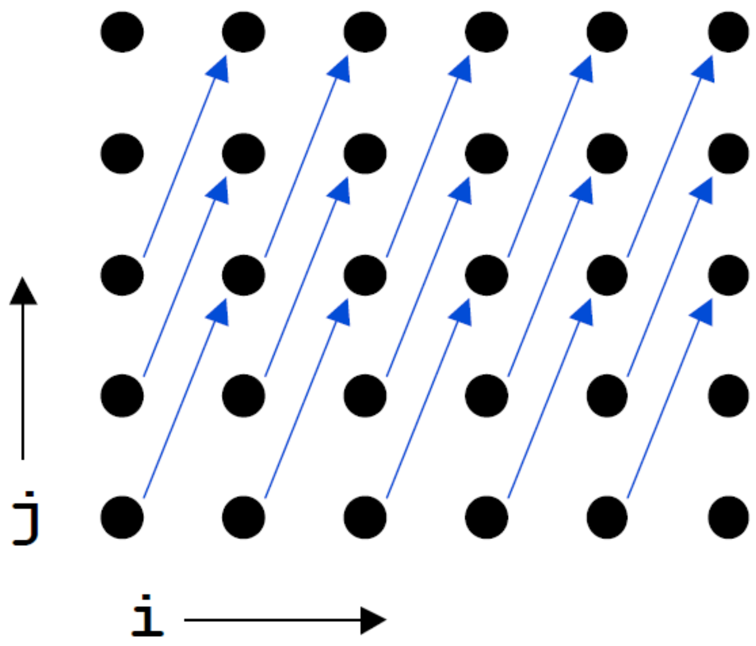
- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

```
do i = 1, 6
  do j = 1, 5
    A(i, j) = A(i-1, j-2) + 1
  enddo
enddo
```

outer loop

- Distance vector: (1, 2)

inner loop



Distance Vectors

- Consider a dependence from statement S1 on iteration i of a loop nest to statement S2 on iteration j
- Dependence distance vector d(i,j) is defined as a vector of length *n* such that $\boldsymbol{d}(\boldsymbol{i},\boldsymbol{j})_k = \boldsymbol{j}_k - \boldsymbol{i}_k$
- Dependence direction vector *D*(*i,j*) is defined as a vector of length *n* such that

$$D(i,j)_k = \begin{cases} - \; if \; D(i,j)_k < 0 \\ 0 \; if \; D(i,j)_k = 0 \\ + \; if \; D(i,j)_k > 0 \end{cases}$$

alternate notations

<	Positive
>	Negative
=	Zero
*	Mixed

Distance Vectors

- Consider a dependence from statement S1 on iteration i of a loop nest to statement S2 on iteration j
- Dependence distance vector d(i,j) is defined as a vector of length *n* suc
- De gth *n* suc

For a valid dependence, the leftmost non-"0" component of the direction vector must be "+"

$$D(i,j)_k = \begin{cases} -\ if\ D(i,j)_k < 0 \\ 0\ if\ D(i,j)_k = 0 \\ +\ if\ D(i,j)_k > 0 \end{cases}$$

alternate notations

<	Positive
>	Negative
=	Zero
*	Mixed

Distance and Direction Vector Example

```
    DO I = 1, N
      DO J = 1, M
S1      A(I,J) = …
S2      … = A(I,J) + …
      ENDDO
    ENDDO
```

```
    DO I = 1, N
      DO J = 1, M
S1      A(I,J) = A(I,1) + …
      ENDDO
    ENDDO
```

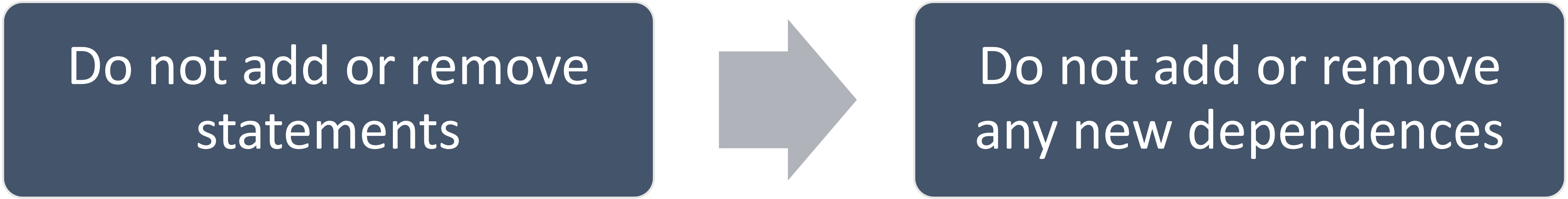
```
    DO I = 1, N
      DO J = 1, M
        DO K = 1, L
S1        A(I+1,J,K-1) = A(I,J,K) + 10
        ENDDO
      ENDDO
    ENDDO
```

```
    FOR I = 1, 5
      FOR J = 1, 5
S1      A(I,J) = A(I,J-3) + A(I-2,J) +
                 A(I-1,J+2) + A(I+1,J-1)
      ENDFOR
    ENDFOR
```

Program Transformations and Validity

Reordering Transformations

- A reordering transformation does not add or remove statements from a loop nest
  - Only reorders the execution of the statements that are already in the loop



Direction Vector Transformation

- Let *T* be a transformation is applied to a loop nest
  - Does not rearrange the statements in the body of the loop
- *T* 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-“0” component that is “-”

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

Dependence Types

- If in a loop, statement S_2 depends on S_1, then there are two possible ways of this dependence occurring
  - S_1 and S_2 execute on different iterations - this is called a **loop-carried** dependence
  - S_1 and S_2 execute on the same iteration - this is called a **loop-independent** dependence

- These types partition all possible data dependences

Loop-Carried Dependences

- S1 can reference the common location on one iteration of a loop; on a subsequent iteration S2 can reference the same location

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

i. S1 references location *M* on iteration i
ii. S2 references *M* on iteration j
iii. d(i,j) > 0 (that is, contains a “+” as leftmost non-“0” component)

Level of Loop-Carried Dependence

- The *level* of a loop-carried dependence is the index of the leftmost non-“0” of *D*(*i,j*) for the dependence.

```
      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
```

Utility of Dependence Levels

- A reordering transformation preserves all level-*k* dependences if it
  i. preserves the iteration order of the level-*k* loop
  ii. does not interchange any loop at level < *k* to a position inside the level-*k* loop and
  iii. does not interchange any loop at level > *k* to a position outside the level-*k* loop.

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

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

Is this transformation valid?

```
  DO I = 1, 10
    DO J = 1, 10
      DO K = 1, 10
S       A(I+1,J+2,K+3) = A(I,J,K) + B
      ENDDO
    ENDDO
  ENDDO
```

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

Loop-Independent Dependences

- S1 and S2 can both reference the common location on the same loop iteration, but with S1 preceding S2 during execution of the loop iteration.

i. S1 refers to memory location *M* on iteration i
ii. S2 refers to *M* on iteration j and i = j
iii. There is a control flow path from S1 to S2 within the iteration.

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

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

Is this transformation valid?

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

```
    D(1) = A(1) + E
    DO I = 2, N
S1:   A(I-1) = B(I-1) + C
S2:   D(I) = A(I) + E
    ENDDO
    A(N) = B(N) + C
```

Dependence Testing

Dependence Testing

- Dependence question
  - Can 4*I be equal to 2*I+1 for I in [1, N] ?

```
DO I=1, N
  A(4*I) = …
  … = A(2*I+1)
ENDDO
```

affine

Given (i) two subscript functions *f* and *g*, and (ii) lower and upper loop bounds L and U respectively, does $f(i_1) = g(i_2)$ have a solution such that $L \leq i_1, i_2 \leq U$?

Multiple Loop Indices, Multi-Dimensional Array

```
DO i=1,n
  DO j=1,m
  X(a1*i + b1*j + c1) = …
  … = X(a2*i + b2*j + c2)
  ENDDO
ENDDO
```

```
DO i=1,n
  DO j=1,m
    X(a1*i1+b1*j1+c1, d1*i1+e1*j1+f1) = …
    … = X(a2*i2+b2*j2+c2, d2*i2+e2*j2+f2)
  ENDDO
ENDDO
```

complex

- Dependence test

$$a_1 * i_1 + b_1 * j_1 + c_1 = a_2 * i_2 + b_2 * j_2 + c_2$$
$$1 \leq i_1, i_2 \leq n$$
$$1 \leq j_1, j_2 \leq m$$

$$a_1 i_1 + b_1 j_1 + c_1 = a_2 i_2 + b_2 j_2 + c_2$$
$$d_1 i_1 + e_1 j_1 + f_1 = d_2 i_2 + e_2 j_2 + f_2$$
$$1 \leq i_1, i_2 \leq n$$
$$1 \leq j_1, j_2 \leq m$$

Complexity in Dependence Testing

- Subscript: A pair of subscript positions in a pair of array references
  - `A(i,j) = A(i,k) + C`
  - `<i,i>` is the first subscript, `<j,k>` is the second subscript
- A subscript is said to be
  - Zero index variable (ZIV) if it contains no index
  - Single index variable (SIV) if it contains only one index
  - Multi index variable (MIV) if it contains more than one index
    - `A(5,I+1,j) = A(1,I,k) + C`
    - First subscript is ZIV, second subscript is SIV, third subscript is MIV

Separability and Coupled Subscript Groups

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

- Coupling indicates complexity in dependence testing

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

Overview of Dependency Testing

1. Partition subscripts of a pair of array references into separable and coupled groups
2. Classify each subscript as ZIV, SIV or MIV
3. For each separable subscript apply single subscript test
   - If not done, go to next step
4. For each coupled group apply multiple subscript test like Delta Test
5. If still not done, merge all direction vectors computed in the previous steps into a single set of direction vectors

https://homes.luddy.indiana.edu/achauhan/Teaching/B629/2006-Fall/LectureNotes/26-dependence-testing.html

Data Dependence Testing with GCD

- Variables in loop indices are integers → Diophantine equations
- The Diophantine equation $a_1 i_1 + a_2 i_2 + \cdots + a_n i_n = c$ has a solution iff $\gcd(a_1, a_2, \ldots, a_n)$ evenly divides c
- If there is a solution, we can test if it lies within the loop bounds. If not, then there is no dependence

```
    for i = 1 to N
S1      a[x*i+k] = …
S2      … = a[y*i+m];
```

x*i1+k = y*i2+m, where $0 \le i_1, i_2 \le N$

- If GCD(x,y) divides (m-k), then a dependence may exist between S1 and S2.
- If GCD(x,y) does not divide (m-k), then S1 and S2 are independent and can be executed at parallel.

Examples:

- `15*i+6*j-9*k=12` has a solution, gcd=3
- `2*i+7*j=3` has a solution, gcd=1
- `9*i+3*j+6*k=5` has no solution, gcd=3

Dependence Testing

```
    for i₁ = L₁:U₁:S₁
      for i₂ = L₂:U₂:S₂
       …
        for iₙ = Lₙ:Uₙ:Sₙ
S1        A[f₁(i₁,i₂,…,iₙ), f₂(i₁,i₂,…,iₙ), …, fₘ(i₁,i₂,…,iₙ)] = …
S2        … = A[g₁(i₁,i₂,…,iₙ), g₂(i₁,i₂,…,iₙ), …, gₘ(i₁,i₂,…,iₙ)]
```

A dependence exists from S1 to S2 if there exists two iteration vectors α and β such that $f_i(\alpha) = g_i(\beta)$, for all i, $1 \leq i \leq m$.

Solving the system of equations for arbitrary functions f and g is undecidable

- The functions can have arbitrary behaviour

Approximate Dependence Testing

- Following system of equations with 2n variables and m equations is the most common

$$
\begin{aligned}
a_{11}i_1+a_{12}i_2+\ldots+a_{1n}i_n+c_1 &= b_{11}j_1+b_{12}j_2+\ldots+b_{1n}j_n+d_1 \\
a_{21}i_1+a_{22}i_2+\ldots+a_{2n}i_n+c_2 &= b_{21}j_1+b_{22}j_2+\ldots+b_{2n}j_n+d_2 \\
\ldots \\
a_{m1}i_1+a_{m2}i_2+\ldots+a_{mn}i_n+c_m &= b_{m1}j_1+b_{m2}j_2+\ldots+b_{mn}j_n+d_m
\end{aligned}
$$

- Solve the system of the form Ax=B for integer solutions
  - A is a $m \times 2n$ matrix and B is a vector of m elements
- Finding solutions to Diophantine equations is NP-hard

Simple Subscript Tests

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

- ZIV test
  - e1 and e2 are constants or loop invariant symbols
  - If e1!=e2, then no dependence exists
- SIV test
  - Strong SIV test: $\langle a*i+c_1, a*i+c_2 \rangle$
    - a,c1,c2 are constants or loop invariant symbols
    - Example: <4i+1, 4i+5>
    - Solution: d=(c2-c1)/a is an integer and $|d| \leq |U_i - L_i|$
  - Weak SIV test: $\langle a_1*i+c_1, a_2*i+c_2 \rangle$
    - a_1, a_2, c1, c2 are constants or loop invariant symbols
    - Example: <4i+1,2i+5> or <i+3,2i>

Weak SIV Test

- Weak zero SIV: `<a₁*i+c₁,c₂>`
  - Solution: $i=(c_2-c_1)/a_1$ is an integer and $|i| \le |U-L|$

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

```
    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: $\langle a*i+c_1, -a*i+c_2 \rangle$
  - Solution: $i=(c_2-c_1)/2a$ is an integer and $|i| \le |U-L|$

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

```
    DO I = 1, (N+1)/2
S1    A(I) = A(N-I+1) + C
    ENDDO
    DO I = (N+1)/2+1, N
S2    A(I) = A(N-I+1) + C
    ENDDO
```

Other Dependence Tests

- GCD test is simple but not accurate
  - It can tell us that there is no solution
- Other tests
  - Banerjee-Wolfe test: widely used test
  - Power Test: improvement over Banerjee test
  - Omega test: “precise” test, most accurate for linear subscripts
  - Range test: handles non-linear and symbolic subscripts
  - many variants of these tests

```
     for i = 1 to 10
S1       a[i] = b[i] + c[i]
S2       d[i] = a[i-100];
```

- GCD is often 1
- Ignores loop bounds

Banerjee-Wolfe Test

- If the total subscript range accessed by *ref1* does not overlap with the range accessed by *ref2*, then *ref1* and *ref2* are independent

```
for (k=0; k < N; k++) {
  c[f(i)] = …;
  … = c[g(j)];
}
```

True: $\exists i, j \in [0, N-1], i \leq j \ \wedge f(i) = g(j)$

Anti: $\exists i, j \in [0, N-1], i > j \ \wedge f(i) = g(j)$

```
DO j=1,100
  a(j) = …
  … = a(j+200)
ENDDO
```

[1:100]
[201:300]

```
DO j=1,100
  a(j) = …
  … = a(j+5)
ENDDO
```

[1:100]
[6:105]

Delta Test

- Notation represents index values at the source and sink

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

- Let source Iteration be denoted by I_0, and sink iteration be denoted by $I_0 + \Delta I$
- Valid dependence implies $I_0 + 1 = I_0 + \Delta I$
- We get $\Delta I = 1 \Rightarrow$ Loop-carried dependence with distance vector (1) and direction vector (+)

Delta Test

```
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: (+,0,-)

Delta Test

- If a loop index does not appear, its distance is unconstrained and its direction is "*"
  - * denotes union of all 3 directions

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

- The direction vector for the dependence is (+, *)
- (*, +) denotes { (+, +), (0, +), (-, +) }
  - (-, +) denotes a level 1 anti-dependence with direction vector (+, -)

Delta Test

- Extract constraints from SIV subscripts and use them for other subscripts

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

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      A(J-I,I+1,J+K) = A(J-I,I,J+K)
    ENDDO
  ENDDO
ENDDO
```

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

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

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

Solving Integer Inequalities

- The loop nest inequalities specify a convex polyhedron
  - A polyhedron is convex if for two points in the polyhedron, all points on the line between them are also in the polyhedron
- Data dependence implies a search for integer solutions that satisfy a set of linear inequalities
  - Integer linear programming is an NP-complete problem
- Steps
  - Use GCD test to check if integer solutions may exist
  - Use simple heuristics to handle typical inequalities
  - Use a linear integer programming solver that uses a branch-and- bound approach based on Fourier-Motzkin elimination for unsolved inequalities

Fourier-Motzkin Elimination

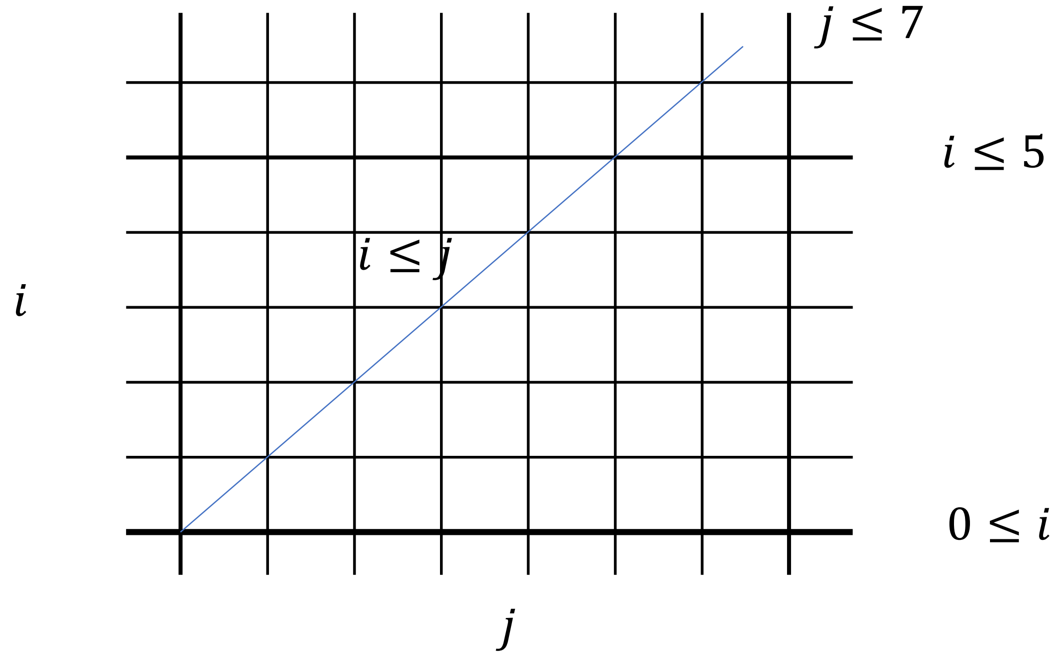
- **INPUT**: an n-dimensional polyhedron S with variables $x_1, x_2, \ldots, x_n$. Eliminate x_m.
- **OUTPUT**: a polyhedron S' with variables $x_1, x_2, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n$
- STEPS
  - Let C be all constraints in S involving x_m

  1. For every pair of a lower bound and upper bound on x_m in C, such as, $L \leq c_1 x_m$ and $c_2 x_m \leq U$, create a new constraint $c_2 L \leq c_1 U$
  2. If integers c_1 and c_2 have a common factor, divide both sides by that factor
  3. If the new constraint is not satisfiable, then there is no solution to S, i.e., S and S' are empty spaces
  4. S' is the set of constraints $S - C$, plus the new constraints generated in Step 2.

Example of Fourier-Motzkin Elimination

- Consider the code

```
for (i = 0; i <= 5; i++)
  for (j = i; j <= 7; j++)
    Z[j,i] = 0;
```



Use Fourier-Motzkin elimination to project the 2D space away from the i dimension and onto the j dimension

$$0 \leq i \wedge i \leq 5 \wedge i \leq j \;\Rightarrow\; 0 \leq j \wedge 0 \leq 5$$

Changing Loop Bounds

```
for (i = 0; i <= 5; i++)
  for (j = i; j <= 7; j++)
    Z[j,i] = 0;
```

⬇

```
for (j = 0; j <= 7; j++)
  for (i = 0; i <= min(5,j); i++)
    Z[j,i] = 0;
```

The constraints are: $0 \leq i, i \leq 5, i \leq j, 0 \leq j, j \leq 7$

Find the loop bounds from the original loop nest.

L_i: 0
U_i: 5, j
L_j: 0
U_j: 7

Use ILP for Dependence Testing

- **Algorithm:**
  - INPUT: A convex polyhedron S, over variables $v_1, v_2, \dots, v_n$
  - OUTPUT: “yes” if S has an integer solution, “no” otherwise

```
for (i=1; i < 10; i++)
  Z[i] = Z[i+10];
```

Show that there are no two dynamic accesses i and i' with $1 \leq i \leq 9$, $1 \leq i' \leq 9$, and $i = i' + 10$.

Dependence Testing is Hard

- Most dependence tests assume affine array subscripts
- Unknown loop bounds can lead to false dependences
- Need to be conservative about aliasing
- Triangular loops adds new constraints

How do we compare N and 10?

```
for (i=0; i< N; i++) {
  a[i] = a[i+10];
}
```

```
for (i=0; i< N; i++) {
  for (j = 0; j < i-1; j++) {
    a[i,j] = a[j,i];
  }
}
```

Add j<i as a new constraint

Why is Dependence Analysis Important?

- Dependence information can be used to drive other important loop transformations
  - Goal is to remove dependences or parallelize in the presence of dependences
  - For example, loop parallelization, loop interchange, loop fusion

- We will see many examples soon

References

- R. Bryant and D. O’Hallaron – Computer Systems: A Programmer’s Perspective.
- R. Allen and K. Kennedy – Optimizing Compilers for Modern Architectures.
- Michelle Strout – CS 553: Compiler Construction, Fall 2007.
- Hugh Leather – Compiler Optimization: Dependence Analysis. 2019.
- Rudolf Eigenmann – Optimizing Optimizing Compilers: Source-to-source (high-level) translation and optimization for multicores.
- P. Gibbons – CS 15-745: Array Dependence Analysis & Parallelization.
- A. Chauhan – B629: Dependence Testing
- Qing Yi – Dependence Testing: Solving System of Diophantine Equations.