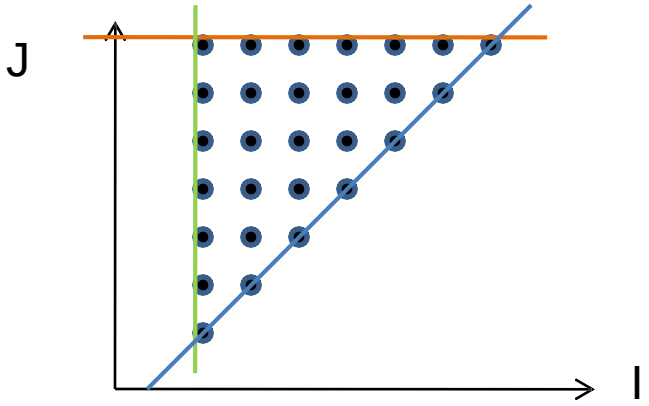
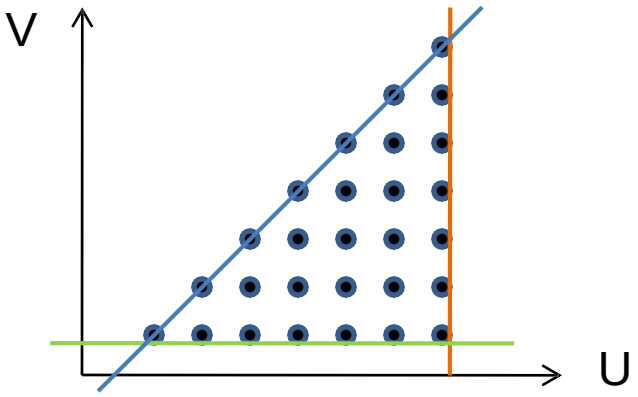
Transformations and Dependences

CS 2210
Spring 2020

Slides modified from Penn State CSE521

Loop permutation can be modeled as a linear transformation on iteration space:






```
Do   I = 1,  N
   Do  J = I,  N
       X(I, J) = 5
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
Do   U= 1,  N
   Do  V = 1,  U
       X(V, U) = 5
```

Permutation of loops in n-loop nest: n x n permutation matrix P

P I = U, where I and U are original and transformed iteration vectors, respectively

Questions:

(1) How do we generate new loop bounds?

(2) How do we modify the loop body?

(3) How do we know when loop interchange is legal?

Code Generation for Transformed Loop Nest

Two problems: (1) Loop bounds (2) Change of variables in body

(1) New bounds:

Original bounds: $A * I \leq b$ where A is in echelon form

Transformation: $U = T * I$

Note: for loop permutation, T is a permutation matrix => inverse is integer matrix

So, bounds on U can be written as $A * T^{-1}U \leq b$

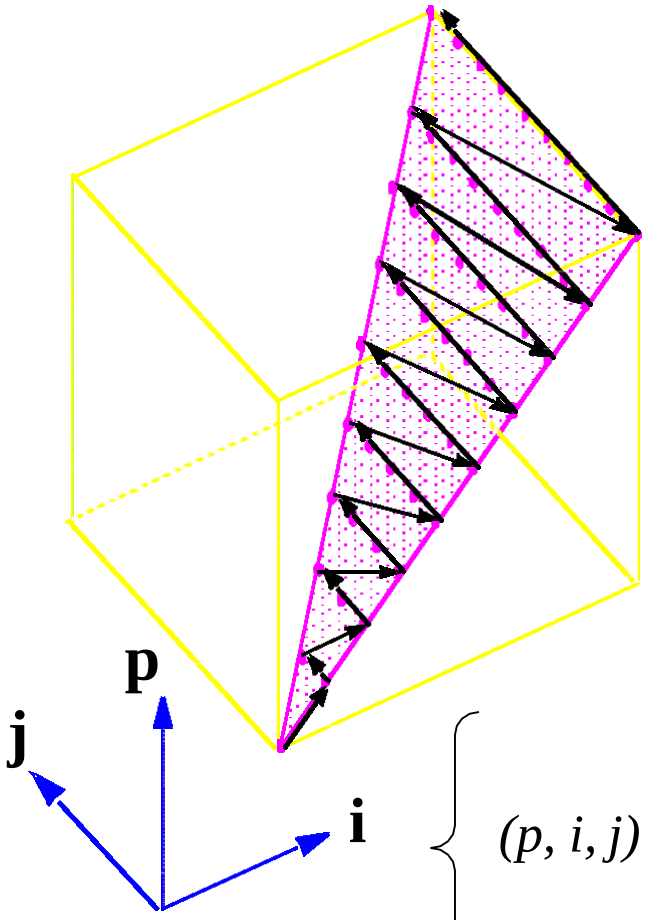
Perform Fourier-Motzkin elimination on this system of inequalities to obtain bounds on U.

(2) Change of variables (fixing loop body):

$I = T^{-1}U$

Replace old variables by new using this formula

Space of Iterations

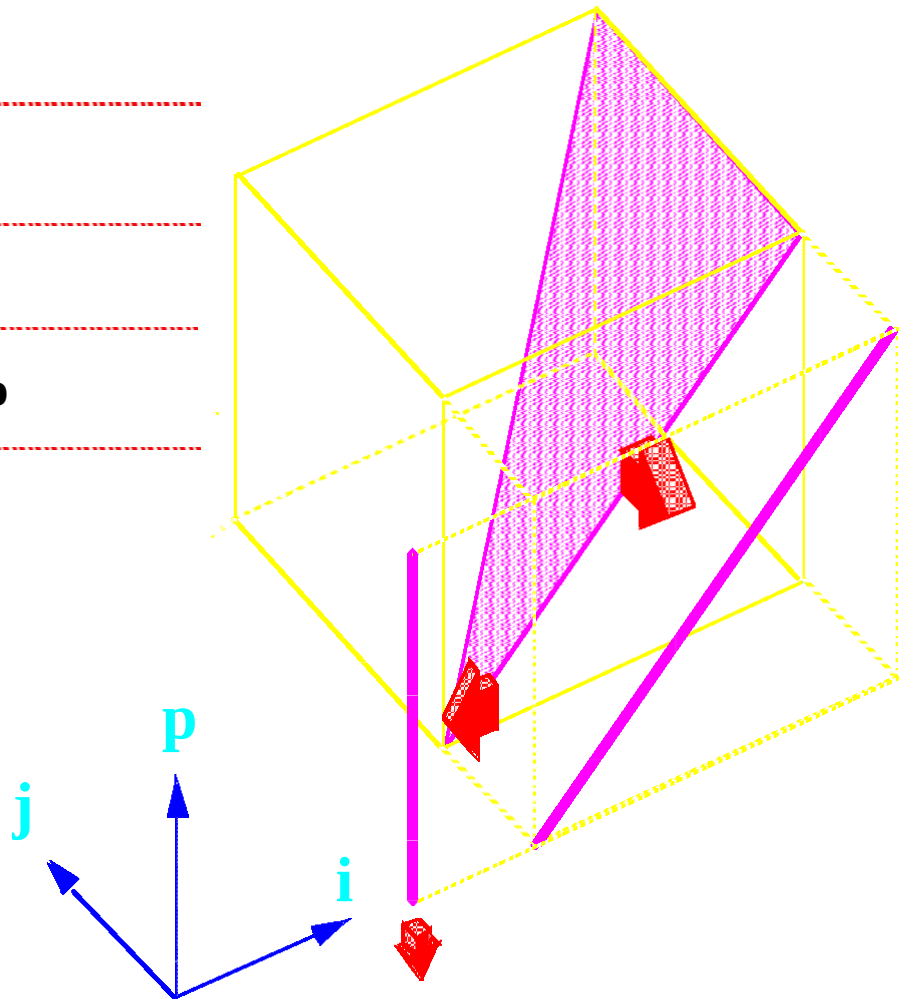


$$\left\{ (p, i, j) \;\middle|\; \begin{array}{c} 2 \le i \le n \\ 1 \le j \le i\text{-}1 \\ i = p \end{array} \right\}$$

```
for p = 2 to n do
  i = p
    for j = 1 to i - 1 do
```

Projections

- **for** p = 2 **to** n **do**
  - i = p
  - **for** j = 1 **to** i - 1 **do**



Fourier Motzkin Elimination

$2 \le i \le n$

$1 \le j \le i-1$

$i = p$

- Project i → j → p

- Find the bounds of i

$2 \le i$

$j+1 \le i$

$p \le i$

$i \le n$

$i \le p$

i: max(2, j+1, p) to min(n, p)

i: p

- Eliminate i

$2 \le n$

$j+1 \le n$

$p \le n$

$2 \le p$

$j+1 \le p$

$p \le p$

$1 \le j$

- Eliminate redundant

$p \le n$

$2 \le p$

$j+1 \le p$

$1 \le j$

- Continue onto finding bounds of j

Further read: https://en.wikipedia.org/wiki/Fourier-Motzkin_elimination

Fourier Motzkin Elimination

$p \leq n$

$2 \leq p$

$j+1 \leq p$

$1 \leq j$

- Find the bounds of j

$1 \leq j$

$j \leq p - 1$

j: 1 to p – 1

- Eliminate j

$1 \leq p - 1$

$p \leq n$

$2 \leq p$

- Eliminate redundant

$2 \leq p$

$p \leq n$

- Find the bounds of p

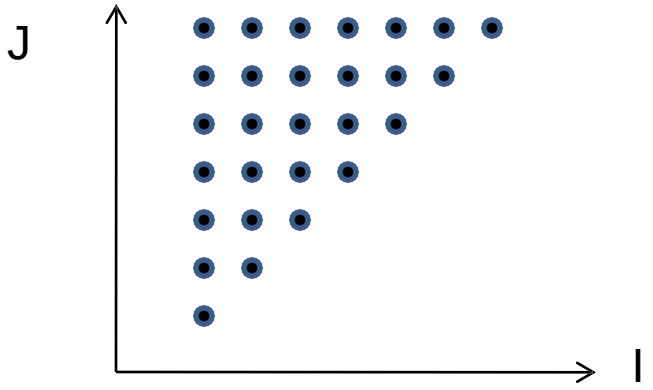
$2 \leq p$

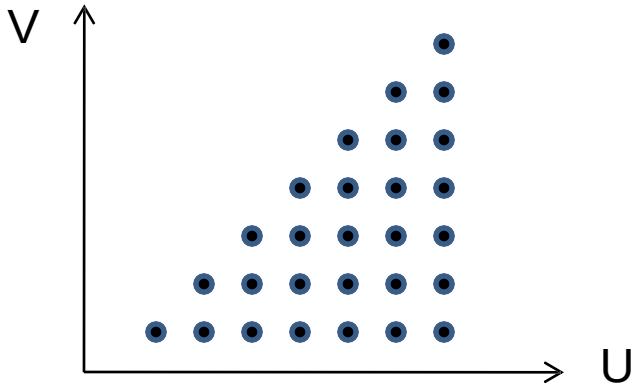
$p \leq n$

p: 2 to n

```
p = my_pid()
if p >= 2 and p <= n then
    for j = 1 to p - 1 do
        i = p
```

Example:






```
Do   I = 1, N
   Do  J = I,  N
       X(I, J) = 5
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
Do   U= 1, N
   Do  V = 1,  min ( U, N)
       X(V, U) = 5
```

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} \le \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} \le \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

Fourier-Motzkin Elimination

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 \\ 0 & 1 \\ -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

Projecting out V from system gives

$1 \leq U \leq N$

Bounds for V are

$1 \leq V \leq \min(U, N)$

These are loop bounds given by FM elimination.

With a little extra work, we can simplify the upper bound of V to U.

Key points:

- Loop bounds determination in transformed code is mechanical.
- Polyhedral algebra technology can handle very general bounds with max’s in lower bounds and min’s in upper bounds.

When is permutation legal?

Position so far: if there is a dependence between iterations, then permutation is illegal.

Do I = 1, 100

Do J = 1, 100

X(2I, J) = ……. X (2I-1, J-1) ……..

Is there a flow dependence between different iterations?

$$1 \le Iw, Ir, Jw, Jr \le 100$$

$$(Iw, Jw) \prec (Ir, Jr)$$

$$2Iw = 2Ir - 1$$

$$Jw = Jr - 1$$

ILP decision problem: is there an integer in union of two convex polyhedra?

No => permutation is legal.

Permutation is legal only if dependence does not exist: too simplistic.

Example:

Do I = 1, 100

Do J = 1, 100

X(I, J) = ……. X (I-1, J-1) ……..

Only dependence is flow dependence:

$$1 \le Iw, Ir, Jw, Jr \le 100$$

$$(Iw, Jw) \prec (Ir, Jr)$$

$$Iw = Ir - 1$$

$$Jw = Jr - 1$$

ILP problem has solution: for example, $(Iw=1, Jw=1, Ir=2, Jr=2)$

Dependence exists but loop interchange is legal!

Point: Existence of dependence is a very "coarse" criterion to determine if interchange is legal.

Additional information about dependence may let us decide whether a given transformation is legal.

To get a handle on all this, let us first define "dependence" precisely.

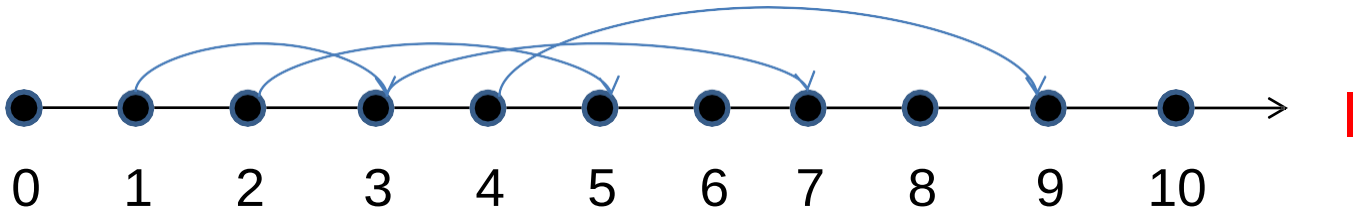
Consider single loop case first:

```
Do I = 1, 100
    X(2I + 1) = …….X(I)……..
```

Flow dependences between iterations:

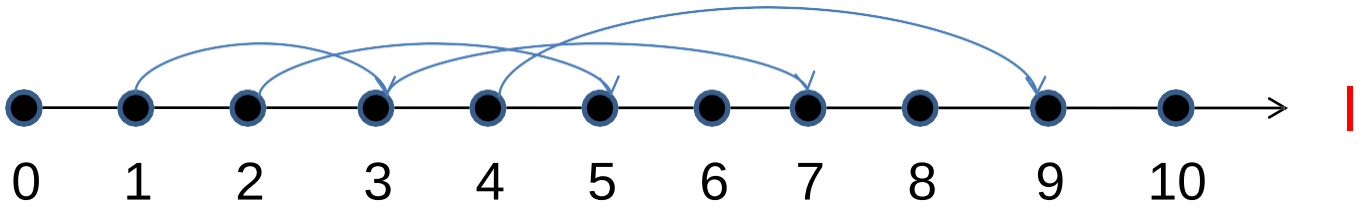
Iteration 1 writes to X(3) which is read by iteration 3.
Iteration 2 writes to X(5) which is read by iteration 5.
……..
Iteration 49 writes to X(99) which is read by iteration 99.

If we ignore the array locations and just think about dependence between iterations, we can draw this geometrically as follows:



Dependence arrows always go forward in iteration space. (e.g., there cannot be a dependence from iteration 5 to iteration 2)

Intuitively, dependence arrows tell us constraints on transformations.



Suppose a transformed program does iteration 2 before iteration 1. **OK!**

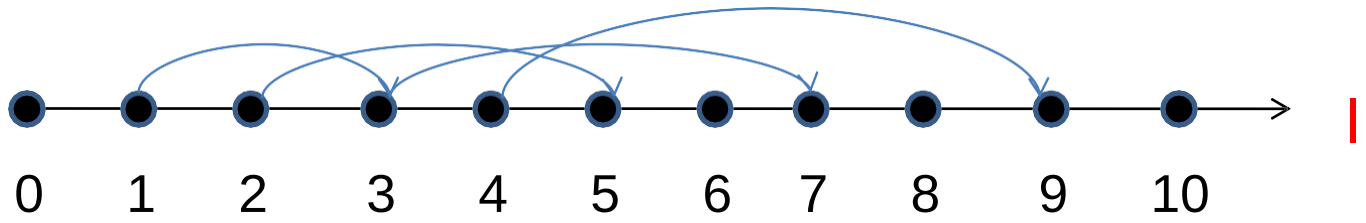
Transformed program does iteration 3 before iteration 1. **Illegal!**

Formal view of a dependence: relation between points in the iteration space.

Do I = 1, 100

X(2I + 1) = …….X(I)…….

Flow dependence = $\{(Iw, 2Iw+1) | 1 \le Iw \le 49\}$



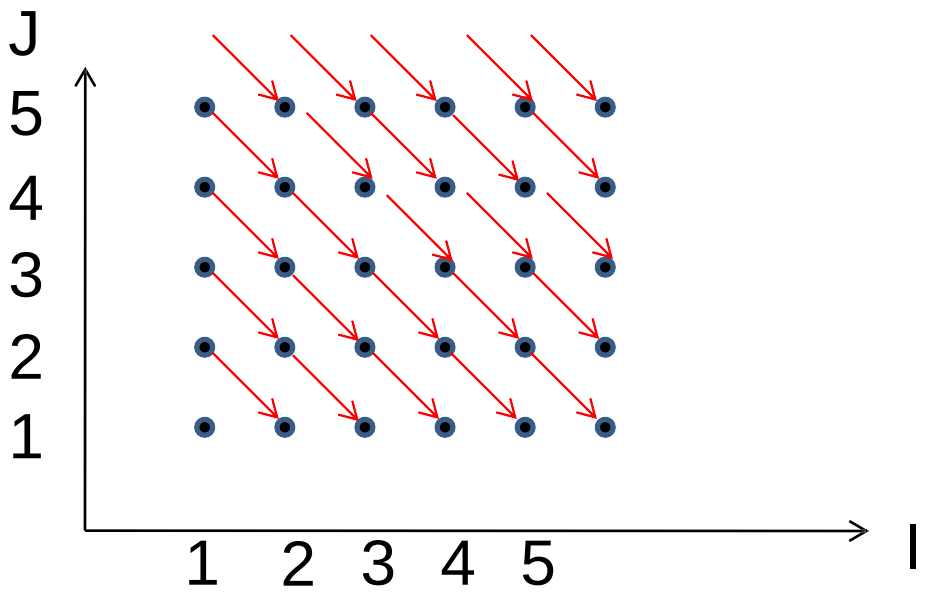
In the spirit of dependence, we will often write this as follows:

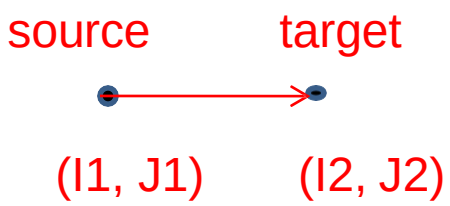
Flow dependence = $\{(Iw \rightarrow 2Iw+1) | 1 \le Iw \le 49\}$

2D loop nest

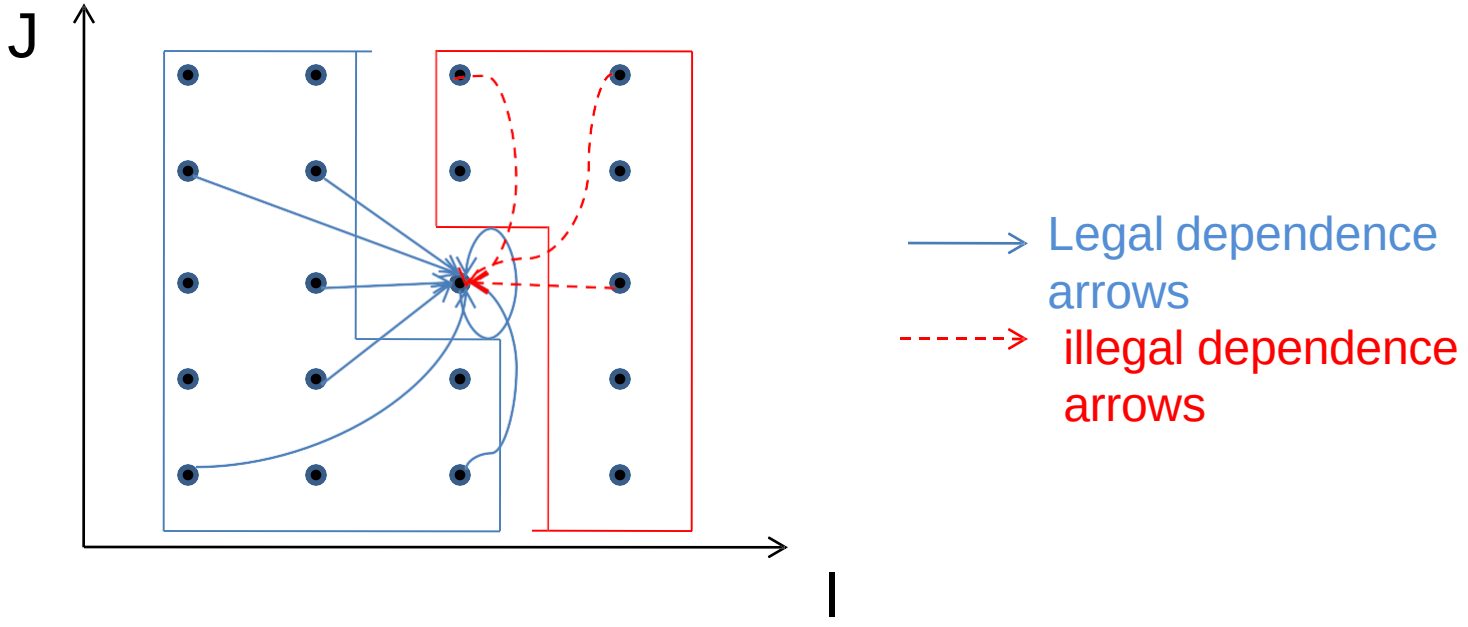
```
Do   10  I  = 1, 100
     Do   10   J = 1, 100
        10   X(I,  J) = X(I-1,  J+1)  + 1
```

Dependence: relation of the form $(I_1, J_1) \rightarrow (I_2, J_2)$.
Picture in iteration space:





Legal and illegal dependence arrows:



If (A → B) is a dependence arrow, then A must be lexicographically less than or equal to B.

Dependence relation can be computed using ILP calculator

```
Do   10  I  = 1, 100
       Do   10   J = 1, 100
          10   X(I,  J) = X(I-1,  J+1)  + 1
```

$$(Iw, Jw) \rightarrow (Ir, Jr)$$

Flow dependence constraints:

$$1 \leq Iw, Ir, Jw, Jr \leq 100$$

$$(Iw, Jw) \prec (Ir, Jr)$$

$$Iw = Ir - 1$$

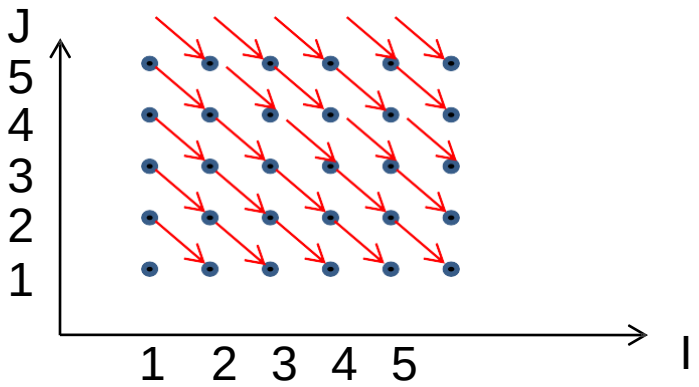
$$Jw = Jr + 1$$

Use ILP calculator to determine the following relation:

$$D = \{(Iw, Jw) \rightarrow (Iw + 1, Jw - 1) \mid (1 \leq Iw \leq 99) \wedge 2 \leq Jw \leq 100)\}$$

If we have the full dependence relation, can we determine when permutation is legal?

Let us look at geometric picture to understand when permutation is legal.

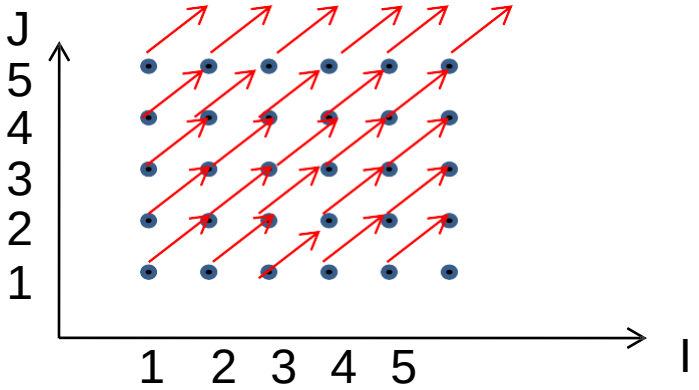


Do I = 1, N

Do J = 1 , N

X(I, J) = X (I-1, J+1).....

Permutation is illegal.



Do I = 1, N

Do J = 1 , N

X(I, J) = X (I-1, J-1).....

Permutation is legal.

Intuitively, if an iteration is dependent on an iteration in its “upper left hand corner”, permutation is illegal. How do we express this formally?

Legality of permutation can be framed as an ILP problem too.

```
Do   10  I  = 1, 100
       Do   10   J = 1, 100
          10   X(I,  J) = X(I-1,  J+1)  + 1
```

Permutation is illegal if there exist iterations in source program such that $(I_1, J_1),(I_2, J_2)$

$((I_1, J_1) \rightarrow (I_2, J_2)) \in D$ (dependent iterations)

$(J_2, I_2) \prec (J_1, I_1)$ (iterations done in wrong order in transformed program)

This can obviously be phrased as an ILP problem and solved.

One solution: $(I_1,\ J_1) = (1,2), \quad (I_2, J_2) = (2,1)$

Thus, interchange is illegal.

General picture:

Permutation is co-ordinate transformation: *U=P*I*
where P is a permutation matrix.
Conditions for legality of transformation:
For each dependence D in loop nest, check that there do not exist iterations I_1 and I_2 such that

$$(\underline{I_1} \rightarrow \underline{I_2}) \in D$$

$$P(\underline{I_2}) \prec P(\underline{I_1})$$

First condition: dependent iterations
Second condition: iterations are done in wrong order in transformed program.
Legality of permutation can be determined by solving a bunch of ILP problems.

Problems with using full dependence sets:

• Expensive (time/space) to compute full relations

•Need to solve ILP problems again to determine legality of permutation

•Symbolic loop bounds (‘N’) require parameterized sets (‘N’ is unbound variable in definition of dependence set)

Dependence abstractions: summary of dependence set D

• Less information than full set of tuples in D

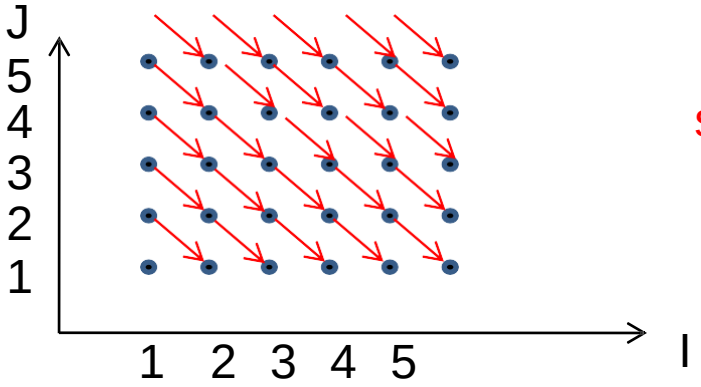
• More information than non-emptiness of D

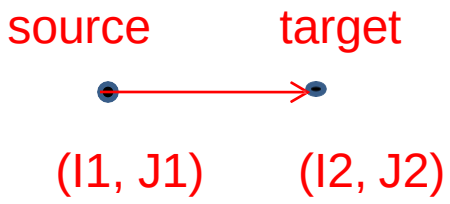
•Intuitively, **“as much as is needed for transformations of interest”**

Distance/direction: Summarize dependence relation

Look at dependence relation from earlier slides:

$$\{(1,2) \rightarrow (2,1), (1,3) \rightarrow (2,2), \ldots (2,2) \rightarrow (3,1) \ldots .\}$$





Difference between dependent iterations = (1, -1). That is, $(Iw, Jw) \rightarrow (Ir, Jr) \in \text{dependencerelation, implies :}$

$$Ir - Iw = 1$$

$$Jr - Jw = -1$$

We will say that the *distance vector* is (1, -1).

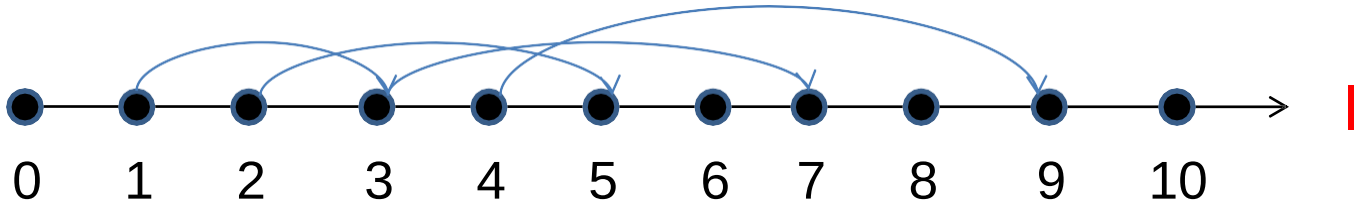
Note: From distance vector, we can easily recover the full relation. In this case, distance vector is an *exact* summary of relation.

Set of dependent iterations usually is represented by many distance vectors.

Do I = 1, 100

X(2I + 1) = …..X(I)…….

Flow dependence = $\{(Iw \rightarrow 2Iw+1) | 1 \leq Iw \leq 49\}$



Distance vectors: {(2), (3), (4), …., (50)}

Distance vectors can obviously never be negative (if (-1) was a distance vector for some dependence, there is an iteration I_1 that depends on iteration I_1+1 which is impossible).

Distance vectors are an approximation of a dependence:

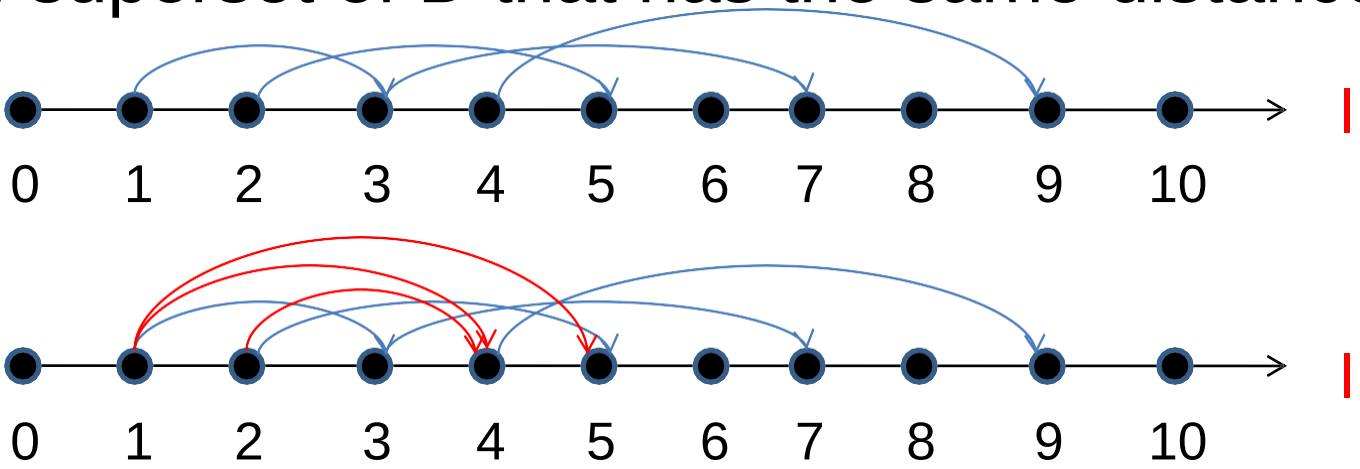
(intuitively, we know the arrows but we do not know their sources.) $D = \{(Iw, 2Iw + 1) \mid 1 \leq Iw \leq 49\}$

Example:

Distance vectors: {(2), (3), (4), ….., (50)}

$$D_1 = \{(I_1, I_2) \mid (1 \leq I_1 \leq 49) \wedge (50 + I_1) \geq I_2 \geq (2I_1 + 1)\}$$

(convex) superset of D that has the same distance vectors.



Both dependences have same set of distance vectors

Computing distance vectors for a dependence

Do I = 1, 100

X(2I + 1) = ……X(I)…….

Flow dependence: $1 \leq Iw \leq Ir \leq 100$

$$2Iw + 1 = Ir$$

Flow dependence = $\{(Iw, 2Iw + 1) \mid 1 \leq Iw \leq 49\}$

Computing distance vectors without computing dependence set:

Introduce a new variable $\Delta = Ir - Iw$ and project onto Δ

$$1 \leq Iw < Ir \leq 100$$

$$2Iw + 1 = Ir$$

$$\Delta = Ir - Iw$$

Solution: $\Delta = \{d \mid 2 \leq d \leq 50\}$

Example: 2D loop nest

```
Do   10   I = 1,  100
     Do    10    J = 1,  100
        10   X(I,  J) = X(I-1, J+1)  + 1
```

Flow dependence constraints: $(Iw, Jw) \rightarrow (Ir, Jr)$

Distance vector: $(\Delta_1, \Delta_2) = (Ir - Iw, Jr - Jw)$

$$1 \leq Iw, Ir, Jw, Jr \leq 100$$

$$(Iw, Jw) \prec (Ir, Jr)$$

$$Iw = Ir - 1$$

$$Jw = Jr + 1$$

$$(\Delta_1, \Delta_2) = (Ir - Iw, Jr - Jw)$$

Solution: $(\Delta_1, \Delta_2) = (1, -1)$

General approach to computing distance vectors:

Set of distance vectors generated from a dependence is itself a polyhedral set.

Computing distance vectors without computing dependence set:

To the linear system representing the existence of the dependence, add new variables corresponding to the entries in the distance vector and project onto these variables.

Reality check:

- In general, dependence is some complicated convex set.
- In general, distance vectors of a dependence are also some complicated convex set!
- What is the point of “summarizing” one complicated set by another equally complicated set?!!

Answer: We use distance vector summary of a dependence only when dependence can be summarized by a single distance vector (called a uniform dependence).

- How do we summarize dependence when we do not have a uniform dependence?

Answer: use direction vectors.