Transformations and Dependences
Recall:

Polyhedral algebra tools for
determining emptiness of convex polyhedra
and enumerating integers in such a polyhedron.

Central ideas:
- reduction of matrices to echelon form by unimodular column operations,
- Fourier-Motzkin elimination

Let us use these tools to determine (i) legality of permutation and (ii) generation of transformed code.
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?

\[ \Omega = \Pi \]  

Permutation of loops in n-loop nest: nxn permutation matrix \( P \)  

\[ \begin{bmatrix} \Lambda \\ \Omega \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \]

Loop permutation can be modeled as a linear transformation on iteration space.
Replace old variables by new using this formula:

$$\overline{N}_{I-L} = \overline{I}$$

Change of variables:

$$\overline{N}$$

inequalities to obtain bounds on system of

Perform Fourier-Motzkin elimination on this system of

So bounds on can be written as $$\overline{L}$$

Note: For loop permutation, $$\overline{L}$$ is a permutation matrix

Transformation:

$$\overline{L} \ast \overline{L} = \overline{L}$$

Original bounds: where $$q \geq \overline{I} \ast \overline{A}$$

New bounds:

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

Code Generation for Transformed Loop Nest
Example:

\[
\begin{bmatrix}
N & 0 & 1 \\
0 & N & 1 \\
1 & 0 & N
\end{bmatrix}
\]

\[
\begin{bmatrix}
\Lambda & \Omega \\
\Omega & \Lambda
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
1 & 0 \\
0 & 1 \\
0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
N & 0 & 1 \\
0 & N & 1 \\
1 & 0 & N
\end{bmatrix}
\]

\[
\begin{bmatrix}
\Lambda \\
\Omega
\end{bmatrix} = \begin{bmatrix}
I \\
I
\end{bmatrix} \begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}
\]

\[
\begin{bmatrix}
\Lambda \\
\Omega
\end{bmatrix} = \begin{bmatrix}
1 \\
1
\end{bmatrix}
\]

\[
\begin{bmatrix}
X \cap \Lambda \\
X \cup \Lambda
\end{bmatrix} = \Lambda
\]

\[
\begin{bmatrix}
X \cup \Lambda \\
X \cap \Lambda
\end{bmatrix} = \Omega
\]

\[
\begin{bmatrix}
X \cup \Lambda \\
X \cap \Lambda
\end{bmatrix}
\]

\[
\begin{bmatrix}
X \cap \Lambda \\
X \cup \Lambda
\end{bmatrix}
\]
With a little extra work, we can simplify the upper bound of $\Lambda$ to $U$.

These are loop bounds given by FM elimination.

Bounds for $\Lambda$ are

\[ \min(U, N) \leq \Lambda \leq 1 \]

Projecting out $\Lambda$ from system gives

\[
\begin{pmatrix}
N & 0 \\
0 & N \\
N & I^- \\
I^- & I^- \\
\end{pmatrix}
\leq
\begin{pmatrix}
0 & 1 \\
1 & 1^- \\
I & 0 \\
I^- & 0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
N & 0 \\
0 & N \\
N & I^- \\
I^- & I^- \\
\end{pmatrix}
\geq
\begin{pmatrix}
0 & 1 \\
1 & 1^- \\
I & 0 \\
I^- & 0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 \\
1^- & 1 \\
0 & 1 \\
0 & 1^- \\
\end{pmatrix}
\]
Key points:

Like:

- No need for pattern matching etc for triangular bounds and the like.
- Polyhedral algebra technology can handle very general bounds.
- Loop bounds determination in transformed code is mechanical.
When is permutation legal? 

If there is dependence between iterations, then permutation is illegal.

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

No permutation is legal.

\[
\begin{align*}
I - J & = mJ \\
I - 2J & = m2J \\
(\text{ILP, ILP}) & \supset (\text{mILP, mILP}) \\
0 \leq mI, mJ & \leq 1
\end{align*}
\]

Is there a flow dependence between different iterations?

\[
\cdots x(2I, j) \cdots = x(2I-1, j-1) \\
\text{DO } j = 1, 100 \\
\text{DO } I = 1, 100
\]

When is permutation illegal?
Dependence exists but loop interchange is illegal

$$\text{ILP problem has solution: for example,} \left( m f, m I \right)$$

$$I - m f = I$$

$$I - m I = I$$

$$\left( m f, m I \right) \succ \left( m f, m I \right)$$

$$\therefore I \geq m f$$

Only dependence is how dependence:

$$\cdots x \left( I - I, f - I \right) = x$$

Do: $$j = I, I \geq 100$$

Do: $$I = I, I \geq 100$$

Example:

Simplistic.

Permutation is legal only if dependence does not exist: too
To get a handle on all this, let is first define dependence precisely:

a transformation is legal.

Additional information about dependence may let us conclude that
determination of interchange is legal.

Point: Existence of dependence is a very "coarse" criterion to
Consider single loop case first:

\[
\begin{align*}
\mathbf{X}(2i+1) &= \mathbf{X}(i) \\
\mathbf{X}(2i+1) &= \mathbf{X}(i) \\
\mathbf{X}(100) &= \mathbf{X}(1) \\
\end{align*}
\]

Do \( i = 1, 2, \ldots, 100 \)

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

Flow dependences between iterations:

Iteration 4 writes to \( \mathbf{X}(99) \) which is read by iteration 99.

Iteration 2 writes to \( \mathbf{X}(9) \) which is read by iteration 5.

Iteration 1 writes to \( \mathbf{X}(3) \) which is read by iteration 3.

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

\[ \{ \forall m \geq m I \geq 1 \mid (I + m I \leftarrow m I) \} = \text{Flow dependence} \]

In the spirit of dependence, we will often write this as follows:
2D loop nest

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

DO 10 J = 1, 100

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

Picture in iteration space:

Step 1: source

Step 2: target
If $(A \rightarrow B)$ is a dependence arrow, then $A$ must be lexicographically less than or equal to $B$. Legal and illegal dependence arrows:

---

Illegal dependence arrows

Illegal and illegal dependence arrows
\{(00 \geq mF \geq 2) \land (99 \geq mI \geq 1)\} \land (I - mF, I + mI) \leftarrow (mF, mI) \equiv D

Use ILP calculator to determine the following relation:

$$I + mF = mI \quad \bullet$$
$$I - mI = mF \quad \bullet$$
$$(mF, mI) \triangleright (mF', mI') \quad \bullet$$
$$00 \geq mF', mF', mI', mI \geq 1 \quad \bullet$$

Flow dependence constraints:

$$0 \leq n \leq 1$$
$$\text{DO to } I = 1, 100$$
$$\text{DO to } J = 1, 100$$

Dependence relation can be computed using ILP calculator.
If we have the full dependence relation, can we determine when permutation is illegal?

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

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

```
DO I = 1,N
  DO J = 1,N
    X(I,J) = X(I-1,J-1)-
```

```
x(I,J) = X(I-1,J+1)
```

Formally, let hand corner, permutation is illegal. How do we express this?
Interchange is illegal.

\((I, I) = (J, J) = (1, J)\).

One solution: \((I, I)\) \(\neq (J, J)\) \(\neq (1, J)\) \(\neq (J, 1)\).

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

\[ \text{(Program)} \]

\begin{align*}
\text{interchange done in wrong order in transformed program}\quad & (I, I) > (J, J) \\
\text{dependent iterations}\quad & (D \in ((I, I) \leftarrow (J, J))) \\
\text{in source program such that}\quad & \text{permutation is illegal if there exist iterations } (I, J) \text{ in source program.}
\end{align*}

\[ 1 + X(I-1, J+1) \]

\[ 10 \text{ DO } 1 \to J = 1, 100 \]

\[ 10 \text{ DO } I = 1, 100 \]

Legality of permutation can be framed as an ILP problem.
General picture:

Permutation is coordinate transformation. Let legality of permutation can be determined by solving a bunch of ILP problems.

Conditions for legality of transformation:

First condition: Dependent iterations

\[(\bar{T})d \succ (\bar{c})d\]

Second condition: Iterations are done in wrong order in transformed program

For each dependence in loop nest, check that there do not exist such that iterations \(\bar{T}\) and \(\bar{c}\) are done in wrong order in transformed program.

Conditions for legality of transformation:

Permutation is coordinate transformation

\[d \cdot \bar{T} = \bar{\nabla}\]

where is a
Problems with using full dependence sets:

- Intuitively, as much as is needed for transformations of interest
  - expensive \( \mathcal{D} \) more information than non-emptiness of
  - \( \mathcal{D} \) less information than full set of tuples in

\( \mathcal{D} \) dependence abstractions: summary of dependence set

- (unbound variable in definition of dependence set
  - \( \mathcal{N} \) requires parameterized sets
  - Symbolic loop bounds require parameterized sets
  - \( \mathcal{N} \) is permutation
  - Need to solve ILP problems again to determine legality of
  - Expensive (time/space) to compute full relations

Problems with using full dependence sets:
In this case, distance vector is an exact summary of relation.

Note: From distance vector, we can easily recover the full relation.

We will say that the distance vector is \([1', -1']\).

\[
I - I' = mI - rI, \\
I = mI - rI
\]

Difference between dependent iterations \([I', -I']\). That is, \([I_r, I_w] \in dfdep\).

Example (I, J):

Distance/direction: Summarize dependence relation.

Look at dependence relation from earlier slides.
a distance $f$ of $I_w$.

Distance vectors:

\[
\{(0, 0), \ldots, (\ell, (\ell+2)), (\ell+1, (\ell+2))\}
\]

Distance vectors can obviously never be negative (if $-1$ was a distance $d$).

\[
\{64 \geq mI \geq 1 | (I + mI \ivalence I)|
\]

Flow dependence $\cdot \cdot \cdot X(I + 1)X(I) = \cdot \cdot \cdot$

Do $I = 1$, 100 vectors.

Set of dependent iterations usually is represented by many distance

\[
I = 1\]
Distance vectors are an approximation of a dependence. We know the arrows but we do not know their sources. Intuitively, we have the same set of distance vectors.

Example: $\mathbf{D} = f(I_1; I_2; I_3; I_4; I_5; I_6; I_7; I_8; I_9)$ is a convex superset of $\mathbf{D}$ that has the same distance vectors.

Both dependences have the same set of distance vectors.

$$\{((\varphi_0), (\varphi_1), (\varphi_2), (\varphi_3), (\varphi_4), (\varphi_5), (\varphi_6), (\varphi_7), (\varphi_8), (\varphi_9)) : (1 + m I_1, \ldots, m I_n) = D\}$$
Computing distance vectors for a dependence DO $I \neq 1$, $I \neq 0$

Flow dependence:

\[ X'(I+1) = X(I) \]

Do $I = 1', 100$

Computing distance vectors without computing dependence set:

Introduce a new variable $mI - \mu I = \nabla$ and project onto $mI - \mu I = \nabla$

\[ \{ 9 \geq mI \geq I \mid (I + mI, mI) \} = \nabla \]

Flow dependence:

\[ \mu I = I + mI \]

\[ 0 \geq \mu I > mI \geq 1 \]
Example: 2D loop nest

\begin{align*}
(1 - \ell) & = (\nabla, \nabla^\top) \\
(\ell f - \ell, \ell I - \ell) & = (\nabla, \nabla^\top) \\
I + \ell f & = \ell f \\
I - \ell I & = \ell I \\
(\ell f, \ell I) & \succ (\ell f, \ell I) \\
0 & \geq \ell f, \ell f, \ell I, \ell I, \ell I > I
\end{align*}
vector and project onto these variables.

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

General approach to computing distance vectors:

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 on to 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 vectors when dependence can be summarized by a single distance vector.

Answer: We use distance vectors 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: We use direction vectors.
\{ \cdots , 4 \leftarrow 2 , 2 \leftarrow 3 , 3 \leftarrow 1 , 1 \leftarrow 100 , 1 \leftarrow 1 , 1 \leftarrow 2 \leftarrow 3 , 3 \leftarrow 1 , 1 \leftarrow 2 \leftarrow 3 , 3 \leftarrow 1 \} \\
\text{Get bigger relation than (1), we have a try to recover dependence relation from direction (+).
}

\text{Direction vectors are not exact.}

(\leq , \leq , \leq ) \text{ or } (\geq , \geq , \geq ) \text{ or } (0) \text{ or } (+) = \text{In general, direction = some distances in range [1, \infty ]}

\text{Intuition: direction } (+) = \text{ direction = some distances in range [1, \infty ]}

\text{Here, direction = } (+) = \text{ direction = some distances in range [1, \infty ]}

\text{But all distances are +ve, so use direction vector instead.}

\text{No fixed distance between dependent iterations!}

\text{Dependence relation: } \{ \cdots , 2 \leftarrow 3 , 3 \leftarrow 2 \leftarrow 3 , 3 \leftarrow 1 , 1 \leftarrow 2 \leftarrow 3 , 3 \leftarrow 1 , 1 \leftarrow 2 \leftarrow 3 , 3 \leftarrow 1 \} \\
\Rightarrow \text{Flow dependence equation: } Z \cdot I = I + m \cdot I \\
\text{Flow dependence equation: } 10 \cdot (Z+1) \cdot X(I) = X(I) + 1 \\
\text{Flow dependence equation: } 10 \cdot (Z+1) \cdot X(I) = X(I) + 1 \\
\text{Do } 10 \cdot I = 1 , 100 \text{ Example: Direction vectors}
Valid dependence vectors are lexicographically positive:

\[
\begin{pmatrix} -1 \\ 0 \\ 1 \\ 2 \\ -2 \\ -3 \\ 3 \\ -1 \\ -3 \\ 0 \\ 1 \\ -1 \\ 2 \\ -2 \\ -3 \end{pmatrix}
\]

The following direction vectors cannot exist:

\[
\begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \\ -1 \\ -2 \\ -3 \\ 0 \\ 1 \\ -1 \\ 2 \\ -2 \\ -3 \end{pmatrix}
\]

Legal direction vectors:

\[
\begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}
\]

Direction: \((I^1 - I^2) \text{ sign} \) \((I^1 - I^2) \text{ sign} \) = \((I^1 - I^2) \text{ sign} \) \((I^1 - I^2) \text{ sign} \)

Distance: \((I^1 - I^2, I^1 - I^2) = \exists \text{ dependence relation, then}

\[
I^1, I^2 \in (I^1, I^2)
\]

Assume loop nest is \((I^1, I^2)\).

Directions for Nested Loops
In a single loop, direction \((-\) cannot occur.

Test for direction \((0)\): add inequality \(\delta I = mI\).

Test for direction \((+)\): add inequality \(\delta I > mI\).

Otherwise, determine the direction(s) of dependence.

If the IP engine says there are no solutions, no dependence.

If any direction (called \((*)\) direction).

First, use inequalities shown above to test if dependence exists in

\[ 00 \gtrsim \delta I \gtrsim 1 \]
\[ 00 \gtrsim mI \gtrsim 1 \]
\[ (\delta I)^j = (mI)^j \]

Focus on how dependences:

\[ \cdots = ((I)^j)X \cdots = 10 \]
\[ \cdots = ((I)^f)X \]

Do \(10\) to \(1\). Use IP engine: \(\bigcup\) Df Directio

How to compute Directions: Use IP engine
Key ideas:

1. Reverse direction vectors top down.

2. Do not test for impossible directions like (−, *) (0, +)

No need to do more tests.

Figure 1: Hierarchical Testing for Nested Loops

Same idea as single loop: Hierarchical testing

Computing Directions of Nested Loops
It is also possible to compute direction vectors by projecting on the iteration difference vector. Similar to what we did for distance vectors, let this as an exercise for you.
Compute dependences for following program:

\[
\begin{align*}
\text{DO } & I = 1, N \\
\text{DO } & J = 1, N
\end{align*}
\]

\[
X(I,J) = \cdots X(I,J) \cdot \cdots
\]

Big hairy example: Compute dependences for following program:
Linear system for anti-dependence:

\[(I - N) \geq 2\mathbf{\nabla} \geq 0\]

\[0 = I\mathbf{\nabla}\]

So directions for anti-dependence are

\[(1 - N) \geq 2\mathbf{\nabla} \geq 0\]

Projecting onto \(\mathbf{\nabla}_1\) and \(\mathbf{\nabla}_2\), we get

\[\left(\mathbf{\nabla}_1 - m\mathbf{\nabla}_1\right) = 2\mathbf{\nabla}\]

\[\left(\mathbf{\nabla}_2 - m\mathbf{\nabla}_2\right) = I\mathbf{\nabla}\]

\[\left(m\mathbf{\nabla}_1 - m\mathbf{\nabla}_2\right) \geq \left(m\mathbf{\nabla}_1 - m\mathbf{\nabla}_2\right)\]

\[N \geq \mathbf{\nabla}_1 - m\mathbf{\nabla}_1 - m\mathbf{\nabla}_2 \geq I\]

\[\mathbf{\nabla}_1 = m\mathbf{\nabla}_1\]

\[\mathbf{\nabla}_2 = m\mathbf{\nabla}_2\]

Linear system for anti-dependence:
Similarly, you can compute direction for flow dependence.

\[
\begin{align*}
+ \\
0 \\
\end{align*}
\]

and also show that no output dependence exists.
In our example, the dependence matrix is:

\[ + \quad 0 \]
\[ 0 \quad 0 \]

Matrix containing all dependence distance/direction vectors for all dependences of loop nest.
Dependence direction/distance are adequate for testing legitimacy of permutation.
If transformation \( P \) is legal and original dependence matrix is \( D \), new dependence matrix is \( T^\top D \).

Check for legality: interchange positions in distance/direction vector & check for lex +ve

\[
\begin{bmatrix}

I_2 - I_1 \\
I_2 - I_1
\end{bmatrix}^T = \begin{bmatrix}
I_1 \\
I_1
\end{bmatrix} - \begin{bmatrix}
I_2 \\
I_2
\end{bmatrix}^T
\]

Distance between iterations = \( I_2 - I_1 \) \( J_2 - J_1 \)

\[
\begin{bmatrix}
I_2 \\
I_2
\end{bmatrix} \rightarrow \begin{bmatrix}
I_1 \\
I_1
\end{bmatrix}
\]

Dependence distance

\[
\begin{bmatrix}
I_2 \\
I_2
\end{bmatrix} = \begin{bmatrix}
I_1 \\
I_1
\end{bmatrix} - \begin{bmatrix}
I_2 \\
I_2
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{Do I = I', N} \\
\text{Do J = I', N}
\end{bmatrix}
\]

\[
\begin{bmatrix}
\vee \\
\wedge
\end{bmatrix} = \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
\vee \\
\wedge
\end{bmatrix} = \begin{bmatrix}
I \\
I
\end{bmatrix}
\]

\[
\begin{bmatrix}
\vee \\
\wedge
\end{bmatrix} = \begin{bmatrix}
J \\
J
\end{bmatrix}
\]
Dependence matrix of transformed program: $D$

$\mathbf{D} \cdot \mathbf{T}$

Matrix in which each column is a distance/direction vector

Dependence matrix: $D$

Transformation matrix: $T$

Correctness of General Permutation

Legality: $\mathbf{T} \cdot \mathbf{D} > 0$
Examples:

\[ \text{Distance vector} = (1',1') \]

\[ \text{Dependence vector of transformed program} = (1',1') \]

\[ X(I', J') = X(I-I', J+1) \]

DO J = 1, N
DO I = 1, N

\[ \text{Distance vector} = (1',1') \]

\[ \text{Dependence vector is legal} \]

\[ \text{Dependence vector is not legal} \]
Remarks on dependence abstractions

A good dependence abstraction for a transformation should have the following properties:

- Easy to compute
- Easy to test for legality
- Easy to determine dependence abstractions for transformed program

Direction vectors are a good dependence abstraction for permutation.
In principle, we can use IP engine to compute all directions.

Reality: most subscripts and loop bounds are simple

First check for simple cases.

Call IP engine for more complex cases.

Engineering a dependence analyzer:
Conclusions

Recent experience: too expensive

Traditional position: exact dependence testing (using IP engine) is

(iii) Cost of exact dependence testing: 3-5% of compile time

(ii) IP engine is called for 3-4% of tests for direction vectors cases (ZIV, strong SIV, weak SIV)

(i) exact dependence testing is OK provided we first check for easy