Overview

- Data and control dependences
- The theory and practice of data dependence analysis
  - K-level loop-carried dependences
  - Loop-independent dependences
- Testing dependence
  - Dependence equations and the polyhedral model
  - Dependence solvers
- Loop transformations
  - Loop parallelization
  - Loop reordering
  - Unimodular loop transformations
Data and Control Dependences

- Fundamental property of all computations

- **Data dependence**: data is produced and consumed in the correct order

- **Control dependence**: a dependence that arises as a result of control flow

<table>
<thead>
<tr>
<th>S₁</th>
<th>PI = 3.14</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₂</td>
<td>R = 5.0</td>
</tr>
<tr>
<td>S₃</td>
<td>AREA = PI * R ** 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S₁</th>
<th>IF (T.EQ.0.0) GOTO S₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₂</td>
<td>A = A / T</td>
</tr>
<tr>
<td>S₃</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>
Data Dependence

Definition

There is a *data dependence* from statement $S_1$ to statement $S_2$ iff

1) both statements $S_1$ and $S_2$ access the same memory location and at least one of them stores into it, and

2) there is a feasible run-time execution path from statement $S_1$ to $S_2$
Data Dependence Classification

- **Definition**

A dependence relation $\delta$ is

- A *true dependence* (or flow dependence), denoted $S_1 \delta S_2$

- An *anti dependence*, denoted $S_1 \delta^{-1} S_2$

- An *output dependence*, denoted $S_1 \delta^o S_2$
Compiling for Parallelism

- Theorem

Any instruction statement reordering transformation that preserves every dependence in a program preserves the meaning of that program

- Bernstein’s conditions for loop parallelism:
  - Iteration $I_1$ does not write into a location that is read by iteration $I_2$
  - Iteration $I_2$ does not write into a location that is read by iteration $I_1$
  - Iteration $I_1$ does not write into a location that is written into by iteration $I_2$
Fortran’s PARALLEL DO

- The **PARALLEL** DO requires that there are no scheduling constraints among its iterations
- When is it safe to convert a **DO** to **PARALLEL DO**?

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

OK
```

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

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

Not OK
```

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

- Whereas the `PARALLEL DO` requires the programmer to check if loop iterations can be executed in parallel, an optimizing compiler must analyze loop dependences for automatic `loop parallelization` and `vectorization`.

- There are multiple levels of granularity of parallelism a programmer or compiler can exploit, for example:
  - *Instruction level parallelism* (ILP)
  - *Synchronous* (e.g. SIMD, also vector and SSE) loop parallelism (fine grain) aim is to parallelize *inner loops* as vectors ops
  - *Asynchronous* (e.g. MIMD, work sharing) loop parallelism (coarse grain) aim is to schedule *outer loops* as tasks
  - *Task parallelism* aim is to run different tasks (threads) in parallel (coarse grain)
Advanced Compiler Technology

- Powerful *restructuring compilers* transform loops into parallel or vector code
- Must analyze data dependences in loop nests besides straight-line code
Synchronous Parallel Machines

- SIMD architecture: simpler (and cheaper) than MIMD
- Programming model: PRAM (parallel random access machine)
- Processors operate in lockstep executing the same instruction on different portions of the data space
- Application should exhibit data parallelism
- Control flow requires masking operations (similar to predicated instructions), which can be inefficient

Example 4x4 processor mesh SIMD architecture: processors execute in lock-step and execute the same instruction per clock
Asynchronous Parallel Computers

- **Multiprocessors**
  - Shared memory is directly accessible by each PE (processing element)

- **Multicomputers**
  - Distributed memory requires message passing between PEs

- **SMPs**
  - Shared memory in small clusters of processors, need message passing across clusters

![Diagram of multiprocessor system with shared memory and message passing]
Valid Transformations

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

Valid:

```
DO I = 1, 3
  A(I+1) = B(I)
  B(I+1) = A(I)
ENDDO
```

Invalid:

```
DO I = 1, 3, -1
  A(I+1) = B(I)
  B(I+1) = A(I)
ENDDO
```
Fundamental Theorem of Dependence

Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
Dependence in Loops

The statement instances $S_1[i]$ for iterations $i = 1, \ldots, N$ represent the loop execution.

We have the following flow dependences:

- $S_1[1] \delta S_1[2]$,
- $S_1[2] \delta S_1[3]$,
- $\ldots$
- $S_1[N-1] \delta S_1[N]$

Statement instances with flow dependences:

\[
\begin{align*}
S_1[1] & : A(2) = A(1) + B(1) \\
S_1[2] & : A(3) = A(2) + B(2) \\
S_1[3] & : A(4) = A(3) + B(3) \\
& \ldots
\end{align*}
\]
Iteration Vector

Definition

Given a nest of $n$ loops, the iteration vector $i$ of a particular iteration of the innermost loop is a vector of integers

$$i = (i_1, i_2, \ldots, i_n)$$

where $i_k$ represents the iteration number for the loop at nesting level $k$

The set of all possible iteration vectors spans an iteration space over loop statement instances $S_j[i]$
Iteration Vector Example

The iteration space of the statement at $S_1$ is the set of iteration vectors {$(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)$}

Example: at iteration $i = (2,1)$ statement instance $S_1[i]$ assigns the value of $A(1,2)$ to $A(2,1)$
Iteration Vector Ordering

- The iteration vectors are naturally ordered according to a lexicographical order
  - For example, iteration (1,2) precedes (2,1) and (2,2)

- Definition

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 \(i_n < j_n\)
Cross-Iteration Dependence

Definition

There exist a dependence from $S_1$ to $S_2$ in a loop nest iff there exist two iteration vectors $i$ and $j$ such that

1) $i < j$ and there is a path from $S_1$ to $S_2$

2) $S_1$ accesses memory location $M$ on iteration $i$ and $S_2$ accesses memory location $M$ on iteration $j$

3) one of these accesses is a write
Dependence Example

- Show that the loop has no cross-iteration dependence

- Answer: there are no iteration vectors $i$ and $j$ in $\{(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)\}$ such that $i < j$ and
  - $S_1$ in $i$ writes to the same element of $A$ that is read at $S_1$ in iteration $j$
  - or $S_1$ in iteration $i$ reads an element $A$ that is written at $S_1$ in iteration $j$

```plaintext
DO I = 1, 3
    DO J = 1, I
        A(I,J) = A(J,I)
    ENDDO
ENDDO
```
Dependence Distance and Direction Vectors

Definition

Given a dependence from $S_1$ on iteration $i$ to $S_2$ on iteration $j$, the dependence distance vector $d(i, j)$ is defined as $d(i, j) = j - i$

Given a dependence from $S_1$ on iteration $i$ and $S_2$ on iteration $j$, the dependence direction vector $D(i, j)$ is defined for the $k^{th}$ component as

$$D(i, j)_k = \begin{cases} 
  "<" & \text{if } d(i, j)_k > 0 \\
  "=" & \text{if } d(i, j)_k = 0 \\
  ">" & \text{if } d(i, j)_k < 0 
\end{cases}$$
Direction of Dependence

- There is a flow dependence if the iteration vector of the write is *lexicographically less than* the iteration vector of the read.

- In other words, if the direction vector is *lexicographically positive*

\[ D(i, j)_k = (\_, \_, \ldots, \_, =, <, \ldots) \]

   All ‘\='    Any ‘<', ‘\=', or ‘>’
Representing Dependences with Data Dependence Graphs

- It is generally infeasible to represent all data dependences that arise in a program.
- Usually only static data dependences are recorded.
  - $S_1 \delta(=) S_2$ means $S_1[i] \delta S_2[i]$ for all $i = 1, \ldots, 10000$
  - $S_2 \delta(<) S_2$ means $S_1[i] \delta S_2[j]$ for all $i, j = 1, \ldots, 10000$ with $i < j$
- A data dependence graph compactly represent data dependences in a loop nest.

```plaintext
DO I = 1, 10000
  $S_1 A(I) = B(I) * 5$
  $S_2 C(I+1) = C(I) + A(I)$
ENDDO
```

Static data dependences for accesses to $A$ and $C$:
- $S_1 \delta(=) S_2$ and $S_2 \delta(<) S_2$
Example 1

```
DO I = 1, 3
  DO J = 1, I
  S1 A(I+1,J) = A(I,J)
  ENDDO
ENDDO
```

Flow dependence between $S_1$ and itself on:

- $i = (1,1)$ and $j = (2,1)$: $d(i,j) = (1,0)$, $D(i,j) = (<, =)$
- $i = (2,1)$ and $j = (3,1)$: $d(i,j) = (1,0)$, $D(i,j) = (<, =)$
- $i = (2,2)$ and $j = (3,2)$: $d(i,j) = (1,0)$, $D(i,j) = (<, =)$
Example 2

\[
\begin{align*}
&\text{DO } I = 1, 4 \\
&\text{DO } J = 1, 4 \\
&S_1 \\
&A(I, J+1) = A(I-1, J) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

Distance vector is (1,1) 

\[
S_1 \, \delta_{(<, <)} \, S_1
\]
Example 3

Distance vector is (1,-1)

\[ S_1 \delta_{(<,>)} S_1 \]

```
DO I = 1, 4
   DO J = 1, 5-I
      A(I+1,J+I-1) = A(I,J+I-1)
   ENDDO
ENDDO
```
Example 4

Distance vectors are (0,1) and (1,0)

$$S_1 \delta_{(=,<)} S_1$$
$$S_2 \delta_{(<,=)} S_1$$
$$S_2 \delta_{(<,=)} S_2$$
Loop-Independent Dependences

Definition

Statement $S_1$ has a \textit{loop-independent dependence} on $S_2$ iff there exist two iteration vectors $i$ and $j$ such that

1) $S_1$ 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

That is, the direction vector contains only ‘=’
K-Level Loop-Carried Dependences

Definition
Statement $S_1$ has a *loop-carried dependence* on $S_2$ iff
1) there exist two iteration vectors $i$ and $j$ such that $S_1$ refers to memory location $M$ on iteration $i$ and $S_2$ refers to $M$ on iteration $j$
2) The direction vector is lexicographically positive that is, $D(i, j)$ contains a “<” as its leftmost non-“=” component

A loop-carried dependence from $S_1$ to $S_2$ is
1) *lexically forward* if $S_2$ appears after $S_1$ in the loop body
2) *lexically backward* if $S_2$ appears before $S_1$ (or if $S_1=S_2$)

The *level* of a loop-carried dependence is the index of the leftmost non-“=” of $D(i, j)$ for the dependence
Example 1

- The loop-carried dependence $S_1 \delta(<) S_2$ is forward.
- The loop-carried dependence $S_2 \delta(<) S_1$ is backward.
- All loop-carried dependences are of level 1, because $D(i, j) = (<)$ for every dependence.

```fortran
DO I = 1, 3
S_1 A(I+1) = F(I)
S_2 F(I+1) = A(I)
ENDDO
```

$S_1 \delta(<) S_2$ and $S_2 \delta(<) S_1$
Example 2

- All loop-carried dependences are of the 3rd level, $D(i, j) = (=, =, <)$

- Level-$k$ dependences are sometimes denoted by $S_x \delta_k S_y$

\[ S_1 \delta_{(=,=,<)} S_1 \]

\[ S_1 \delta_3 S_1 \quad \text{Alternative notation for a level-3 dependence} \]
Combining Direction Vectors

A loop nest can have multiple different directions at the same loop level \( k \)

We abbreviate a level-\( k \) direction vector component with "*" to denote any direction
How to Determine Dependences?

- A system of *Diophantine dependence equations* is set up and solved to test direction of dependence (flow/anti)
  - Proving there is (no) solution to the equations means there is (no) dependence
  - Most dependence solvers solve Diophantine equations over affine array index expressions of the form
    \[ a_1 i_1 + a_2 i_2 + \ldots + a_n i_n + a_0 \]
    where \( i_j \) are loop index variables and \( a_k \) are integer

- Non-linear subscripts are problematic
  - Symbolic terms
  - Function values
  - Indirect indexing
  - etc
Dependence Equation: Testing Flow Dependence

To determine flow dependence: prove the dependence equation 
\[ f(\alpha) = g(\beta) \]
has a solution for iteration vectors \( \alpha \) and \( \beta \), such that \( \alpha < \beta \)

\[
\begin{align*}
\text{DO } I & = 1, N \\
S_i & \quad A(I) = A(\beta) \\
\text{ENDDO}
\end{align*}
\]

\[ \alpha + 1 = \beta \] has solution \( \alpha = \beta - 1 \)

\[
\begin{align*}
\text{DO } I & = 1, N \\
S_i & \quad A(I+1) = A(I) \\
\text{ENDDO}
\end{align*}
\]

\[ 2\alpha + 1 = 2\beta \] has no solution

\[
\begin{align*}
\text{DO } I & = 1, N \\
S_i & \quad A(2*I+1) = A(2*I) \\
\text{ENDDO}
\end{align*}
\]
dependence equation: testing anti dependence

\[\begin{align*}
\text{DO } & I = 1, N \\
S_1 & \text{ A(}f(I)\text{) = A(}g(I)\text{)} \\
\text{ENDDO}
\end{align*}\]

To determine anti dependence: prove the dependence equation
\[f(\alpha) = g(\beta)\]
has a solution for iteration vectors \(\alpha\) and \(\beta\), such that \(\beta < \alpha\)

\[\begin{align*}
\text{DO } & I = 1, N \\
S_1 & \text{ A(I+1) = A(I)} \\
\text{ENDDO}
\end{align*}\]

\[\alpha + 1 = \beta\] has no solution

\[\begin{align*}
\text{DO } & I = 1, N \\
S_1 & \text{ A(2*I) = A(2*I+2)} \\
\text{ENDDO}
\end{align*}\]

\[2\alpha = 2\beta + 2\] has solution \(\alpha = \beta + 1\)
Dependence Equation Examples

To determine **flow dependence**:
prove there are iteration vectors $\alpha < \beta$ such that $f(\alpha) = g(\beta)$

\[
\begin{align*}
\text{DO } & I = 1, N \\
& \text{DO } J = 1, N \\
S_i & \quad A(I-1) = A(J) \\
& \text{ENDDO} \\
& \text{ENDDO}
\end{align*}
\]

$\alpha = (\alpha_I, \alpha_J)$ and $\beta = (\beta_I, \beta_J)$
$\alpha_I - 1 = \beta_J$ has solution

\[
\begin{align*}
\text{DO } & I = 1, N \\
S_i & \quad A(5) = A(I) \\
& \text{ENDDO}
\end{align*}
\]

$5 = \beta$ is solution if $5 < N$

\[
\begin{align*}
\text{DO } & I = 1, N \\
S_i & \quad A(5) = A(6) \\
& \text{ENDDO}
\end{align*}
\]

$5 = 6$ has no solution
ZIV, SIV, and MIV

- ZIV (zero index variable) subscript pairs
- SIV (single index variable) subscript pairs
- MIV (multiple index variable) subscript pairs

```plaintext
DO I = ...
  DO J = ...
    DO K = ...
      S1 A(5, I+1, J) = A(N, I, K)
    ENDDO
  ENDDO
ENDDO

ZIV pair  SIV pair  MIV pair
```
Example

- First distance vector component is easy (SIV), but second is not so easy (MIV)

\[ S_1 \delta_{\langle,\rangle} S_1 \]
Separability and Coupled Subscripts

- When testing multidimensional arrays, subscript are separable if indices do not occur in other subscripts.
- ZIV and SIV subscripts pairs are separable.
- MIV pairs may give rise to coupled subscript groups.

```plaintext
DO I = ...
  DO J = ...
    DO K = ...
      A(I,J,J) = A(I,J,K)
  ENDDO
ENDDO
ENDDO
```

SIV pair is separable
MIV pair gives coupled indices
Dependence Testing Overview

1. Partition subscripts into separable and coupled groups
2. Classify each subscript as ZIV, SIV, or MIV
3. For each separable subscript, apply the applicable single subscript test (ZIV, SIV, or MIV) to prove independence or produce direction vectors for dependence
4. For each coupled group, apply a multiple subscript test
5. If any test yields independence, no further testing needed
6. Otherwise, merge all dependence vectors into one set
Loop Normalization

- Some tests use normalized loop iteration spaces
  - Set lower bound to zero (or one) by adjusting limits and occurrence of loop variable
  - For an arbitrary loop with loop index $I$ and loop bounds $L$ and $U$ and stride $S$, the normalized iteration number is $i = (I-L)/S$

- A normalized dependence system consists of a dependence equation along with a set of normalized constraints
Dependence System: Formulating Flow Dependence

- Assuming normalized loop iteration spaces
- A dependence system consists of a dependence equation along with a set of constraints:
  - Solution must lie within loop bounds
  - Solution must be integer
  - Need dependence distance or direction vector (flow/anti)

\[ \alpha < \beta \text{ such that } f(\alpha) = g(\beta) \]

where \( \alpha = (\alpha_I, \alpha_J) \) and \( \beta = (\beta_I, \beta_J) \)

\[
\begin{align*}
\text{DO } i &= 0, 98 \\
\quad \text{DO } j &= 0, i+2-2 \\
S_i &\quad A(i+2,j+1) = A(j+1,i+2) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\alpha_I + 2 &= \beta_J + 1 \\
\alpha_J + 1 &= \beta_I + 2 \\
0 &\leq \alpha_I, \beta_I \leq 98 \\
0 &\leq \alpha_J \leq \alpha_I \\
0 &\leq \beta_J \leq \beta_I \\
\alpha_I &< \beta_I
\end{align*}
\]

\{ \text{Dependence equations} \} \\
\{ \text{Loop constraints} \} \\
\{ \text{Constraint for (<,*) dep. direction} \}
**Dependence System: Matrix Notation**

- **The polyhedral model:** solving a linear (affine) dependence system in matrix notation
  - Polyhedral \( \{ x : Ax \leq c \} \) for some matrix \( A \) and bounds vector \( c \)
  - \( A \) contains the loop constraints (inequalities)
  - Rewrite dependence equation as set of inequalities: \( Ax = b \Rightarrow Ax \leq b \) and \(-Ax \leq -b\)
- If no point lies in the polyhedral (a polytope) then there is no solution and thus there is no dependence

\[
\begin{bmatrix}
1 & 0 & 0 & -1 \\
0 & -1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
\alpha_I \\
\beta_I \\
\alpha_J \\
\beta_J
\end{bmatrix}
= 
\begin{bmatrix}
-1 \\
1
\end{bmatrix}
\]

\[
\begin{bmatrix}
-1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
-1 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 \\
0 & -1 & 0 & 1 \\
1 & -1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\alpha_I \\
\beta_I \\
\alpha_J \\
\beta_J
\end{bmatrix}
\leq
\begin{bmatrix}
0 \\
98 \\
0 \\
98 \\
0 \\
0 \\
0 \\
0 \\
-1
\end{bmatrix}
\]
Fourier-Motzkin Projection

System of linear inequalities
\[ Ax \leq b \]

Projections on \( x_1 \) and \( x_2 \)

\[
\begin{pmatrix}
0 & 1 \\
1 & 1 \\
1 & -1 \\
-2 & -1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2
\end{pmatrix}
\leq
\begin{pmatrix}
6 \\
9 \\
5 \\
-7
\end{pmatrix}
\]
Fourier-Motzkin Variable Elimination (FMVE)

**FMVE procedure:**

1. Select an unknown $x_j$
2. $L = \{i \mid a_{ij} < 0\}$
3. $U = \{i \mid a_{ij} > 0\}$
4. if $L = \emptyset$ or $U = \emptyset$ then $x_j$ is unconstrained (delete it)
5. for $i \in L \cup U$
   
   \[
   A_{[i]} := A_{[i]} / |a_{ij}|
   \]
   
   $b_i := b_i / |a_{ij}|$

6. for $i \in L$
   
   for $k \in U$
   
   add new inequality
   
   \[
   A_{[i]} + A_{[k]} \leq b_i + b_k
   \]

7. Delete old rows $L$ and $U$

Select $x_2$: $L = \{3,4\}$, $U = \{1,2\}$

new system:

\[
\begin{pmatrix}
1 & 0 \\
2 & 0 \\
-2 & 0 \\
-1 & 0
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2
\end{pmatrix}
\leq
\begin{pmatrix}
11 \\
14 \\
-1 \\
2
\end{pmatrix}
\]

\[
\max(1/2,-2) \leq x_1 \leq \min(11,7)
\]
ZIV Test

- Simple and quick test
- The ZIV test compares two subscripts
- If the expressions are proved unequal, no dependence can exist

```
DO I = 1, N
  S_1  A(5) = A(6)
ENDDO

K = 10
DO I = 1, N
  S_1  A(5) = A(K)
ENDDO

K = 10
DO I = 1, N
  DO J = 1, N
    S_1  A(I,5) = A(I,K)
  ENDDO
ENDDO
```
Strong SIV Test

- Requires subscript pairs of the form $aI+c_1$ and $aI'+c_2$ for the def $I$ and use $I'$, respectively.

- Dependence equation $aI+c_1 = aI'+c_2$ has a solution if the dependence distance $d = (c_1 - c_2)/a$ is integer and $|d| \leq U - L$ with $L$ and $U$ loop bounds.

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

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

DO I = 1, N
  S1 A(I+N) = A(I)
ENDDO
```
Weak-Zero SIV Test

- Requires subscript pairs of the form $a_1 I + c_1$ and $a_2 I' + c_2$ with either $a_1 = 0$ or $a_2 = 0$
- If $a_2 = 0$, the dependence equation $I = \frac{(c_2 - c_1)}{a_1}$ has a solution if $(c_2 - c_1)/a_1$ is integer and $L \leq \frac{(c_2 - c_1)}{a_1} \leq U$
- If $a_1 = 0$, similar case

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

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

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

- Requires that \( f(\alpha) \) and \( g(\beta) \) are affine:
  
  \[
  f(\alpha) = a_0 + a_1 \alpha_1 + \ldots + a_n \alpha_n
  \]
  
  \[
  g(\beta) = b_0 + b_1 \beta_1 + \ldots + b_n \beta_n
  \]

- Reordering gives linear Diophantine equation:
  
  \[
  a_1 \alpha_1 - b_1 \beta_1 + \ldots + a_n \alpha_n - b_n \beta_n = b_0 - a_0
  \]

  which has a solution if \( \gcd(a_1, \ldots, a_n, b_1, \ldots, b_n) \) divides \( b_0 - a_0 \)

Which of these loops have dependences?

\[
\begin{align*}
&\text{DO I = 1, N} \\
&S_1 \quad A(2*I+1) = A(2*I) \\
&\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO I = 1, N} \\
&S_1 \quad A(4*I+1) = A(2*I+3) \\
&\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO I = 1, 10} \\
&\quad \text{DO J = 1, 10} \\
&S_1 \quad A(1+2*I+20*J) = A(2+20*I+2*J) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]
Banerjee Test (1)

- $f(\alpha)$ and $g(\beta)$ are affine:
  
  $f(x_1,x_2,\ldots,x_n) = a_0 + a_1 x_1 + \ldots + a_n x_n$
  
  $g(y_1,y_2,\ldots,y_n) = b_0 + b_1 y_1 + \ldots + b_n y_n$

- Dependence equation
  
  $a_0 - b_0 + a_1 x_1 - b_1 y_1 + \ldots + a_n x_n - b_n y_n = 0$

has no solution if 0 is not within the lower bound $L$ and upper bound $U$ of the equation’s LHS.

```plaintext
IF M > 0 THEN
  DO I = 1, 10
  S_1 A(I) = A(I+M) + B(I)
  ENDDO

Dependence equation
  $\alpha = \beta + M$

is rewritten as
  $-M + \alpha - \beta = 0$

Constraints:
  $1 \leq M \leq \infty$
  $0 \leq \alpha \leq 9$
  $0 \leq \beta \leq 9$
```
Banerjee Test (2)

- Test for (*) dependence: possible dependence because 0 lies within $[-\infty,8]$

<table>
<thead>
<tr>
<th>$L$</th>
<th>$U$</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-M + \alpha - \beta$</td>
<td>$-M + \alpha - \beta$</td>
<td>original eq.</td>
</tr>
<tr>
<td>$-M + \alpha - 9$</td>
<td>$-M + \alpha - 0$</td>
<td>eliminate $\beta$</td>
</tr>
<tr>
<td>$-M - 9$</td>
<td>$-M + 9$</td>
<td>eliminate $\alpha$</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$8$</td>
<td>eliminate $M$</td>
</tr>
</tbody>
</table>

Constraints:

- $1 \leq M \leq \infty$
- $0 \leq \alpha \leq 9$
- $0 \leq \beta \leq 9$
Banerjee Test (3)

- Test for (<) dependence: assume that $\alpha \leq \beta - 1$
- No dependence because 0 does not lie within $[-\infty, -2]$

<table>
<thead>
<tr>
<th>$L$</th>
<th>$U$</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>$- M + \alpha - \beta$</td>
<td>$- M + \alpha - \beta$</td>
<td>original eq.</td>
</tr>
<tr>
<td>$- M + \alpha - 9$</td>
<td>$- M + \alpha - (\alpha + 1)$</td>
<td>eliminate $\beta$</td>
</tr>
<tr>
<td>$- M - 9$</td>
<td>$- M - 1$</td>
<td>eliminate $\alpha$</td>
</tr>
<tr>
<td>$- \infty$</td>
<td>$- 2$</td>
<td>eliminate $M$</td>
</tr>
</tbody>
</table>

Constraints:

$1 \leq M \leq \infty$
$0 \leq \alpha \leq \beta - 1 \leq 8$
$1 \leq \alpha + 1 \leq \beta \leq 9$
Banerjee Test (4)

- Test for (>)
  dependence: assume that $\alpha + 1 \geq \beta$

- Possible dependence because 0 lies within $[-\infty, 8]$

\[
\begin{align*}
\text{IF } M > 0 \text{ THEN} \\
\text{DO } I = 1, 10 \\
S_1 & \quad A(I) = A(I+M) + B(I) \\
\text{ENDDO}
\end{align*}
\]

\[- M + \alpha - \beta = 0\]

<table>
<thead>
<tr>
<th>$L$</th>
<th>$U$</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>$- M + \alpha - \beta$</td>
<td>$- M + \alpha - \beta$</td>
<td>original eq.</td>
</tr>
<tr>
<td>$- M + \alpha - (\alpha+1)$</td>
<td>$- M + \alpha$</td>
<td>eliminate $\beta$</td>
</tr>
<tr>
<td>$- M - 1$</td>
<td>$- M + 9$</td>
<td>eliminate $\alpha$</td>
</tr>
<tr>
<td>$- \infty$</td>
<td>8</td>
<td>eliminate $M$</td>
</tr>
</tbody>
</table>

Constraints:

\[
\begin{align*}
1 \leq M \leq \infty \\
1 \leq \beta - 1 \leq \alpha \leq 9 \\
0 \leq \beta \leq \alpha + 1 \leq 8
\end{align*}
\]
Banerjee Test (5)

- **Test for (=) dependence:**
  - assume that $\alpha = \beta$

- **No dependence because 0 does not lie within $[-\infty,-1]$**

<table>
<thead>
<tr>
<th>$L$</th>
<th>$U$</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>$- M + \alpha - \beta$</td>
<td>$- M + \alpha - \beta$</td>
<td>original eq.</td>
</tr>
<tr>
<td>$- M + \alpha - \alpha$</td>
<td>$- M + \alpha - \alpha$</td>
<td>eliminate $\beta$</td>
</tr>
<tr>
<td>$- M$</td>
<td>$- M$</td>
<td>eliminate $\alpha$</td>
</tr>
<tr>
<td>$- \infty$</td>
<td>$- 1$</td>
<td>eliminate $M$</td>
</tr>
</tbody>
</table>

Constraints:

- $1 \leq M \leq \infty$
- $0 \leq \alpha = \beta \leq 9$
Testing for All Direction Vectors

- Refine dependence between a pair of statements by testing for the most general set of direction vectors and upon failure refine each component recursively.

\[
\begin{array}{ccc}
(*,*,*) & (=,*,*) & (>,* ,*) \\
(*,*,*) & yes (may depend) & \\
(<,*,*) & (=,*,*) & (>,* ,*) \\
yes & & \\
(<,<,*) & (<,=,*) & (<,>,*) \\
no (indep) & yes & \\
(<,=,<) & (<,=,) & (<,=,>) \\
\end{array}
\]
Direction Vectors and Reordering Transformations (1)

- Definitions
  - A reordering transformation is any program transformation that changes the execution order of the code, without adding or deleting any statement executions.
  - A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

- Theorem
  - A reordering transformation is valid if, after it is applied, the new dependence vectors are lexicographically positive.
    - None of the new direction vectors has a leftmost non-“=” component that is “>”
Direction Vectors and Reordering Transformations (2)

A reordering transformation preserves all level-$k$ dependences if

1) it preserves the iteration order of the level-$k$ loop,

2) if it does not interchange any loop at level $< k$ to a position inside the level-$k$ loop, and

3) if it does not interchange any loop at level $> k$ to a position outside the level-$k$ loop
Examples

\begin{align*}
\text{DO } I = 1, 10 \\
S_1 \quad F(I+1) = A(I) \\
S_2 \quad A(I+1) = F(I) \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } I = 1, 10 \\
\text{DO } J = 1, 10 \\
\text{DO } K = 1, 10 \\
S_1 \quad A(I+1,J+2,K+3) = A(I,J,K) + B \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}

Both $S_1 \delta_{(,<)} S_2$ and $S_2 \delta_{(,<)} S_1$ are level-1 dependences

\begin{align*}
\text{DO } I = 1, 10 \\
S_2 \quad A(I+1) = F(I) \\
S_1 \quad F(I+1) = A(I) \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } I = 1, 10 \\
\text{DO } K = 10, 1, -1 \\
\text{DO } J = 1, 10 \\
S_1 \quad A(I+1,J+2,K+3) = A(I,J,K) + B \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{Find deps.}
\end{align*}

\begin{align*}
\text{Choose transform}
\end{align*}

\begin{align*}
\text{Find deps.}
\end{align*}

\begin{align*}
\text{Choose transform}
\end{align*}
Direction Vectors and Reordering Transformations (3)

- A reordering transformation preserves a loop-independent dependence between statements \( S_1 \) and \( S_2 \) if it does not move statement instances between iterations and preserves the relative order of \( S_1 \) and \( S_2 \).
Examples

\[
\text{DO } I = 1, 10 \\
S_1 \quad A(I) = \ldots \\
S_2 \quad \ldots = A(I) \\
\text{ENDDO}
\]

\[
\text{Find deps.}
\]

\[S_1 \delta(=) S_2\text{ is a loop-independent dependence}\]

\[
\text{Choose transform}
\]

\[
\text{DO } I = 10, 1, -1 \\
S_2 \quad A(I) = \ldots \\
S_1 \quad \ldots = A(I) \\
\text{ENDDO}
\]

\[
\text{DO } I = 1, 10 \\
\text{DO } J = 1, 10 \\
S_1 \quad A(I,J) = \ldots \\
S_2 \quad \ldots = A(I,J) \\
\text{ENDDO} \\
\text{ENDDO}
\]

\[
\text{Find deps.}
\]

\[S_1 \delta(=,=) S_1\text{ is a loop-independent dependence}\]

\[
\text{Choose transform}
\]

\[
\text{DO } J = 10, 1, -1 \\
\text{DO } I = 10, 1, -1 \\
S_1 \quad A(I,J) = \ldots \\
S_2 \quad \ldots = A(I,J) \\
\text{ENDDO} \\
\text{ENDDO}
\]
Loop Fission

- Loop fission (or *loop distribution*) splits a single loop into multiple loops
- Enables vectorization
- Enables parallelization of separate loops if original loop is sequential
- Loop fission must preserve all dependence relations of the original loop
Loop Fission: Algorithm

- Compute the *acyclic condensation* of the dependence graph to find a legal order of the loops

```
S_1: DO I = 1, 10
S_2: A(I) = A(I) + B(I-1)
S_3: B(I) = C(I-1)*X + Z
S_4: C(I) = 1/B(I)
S_5: D(I) = sqrt(C(I))
S_6: ENDDO
```

Dependencies:
- \( S_3 \delta(\leq) S_2 \)
- \( S_4 \delta(\leq) S_3 \)
- \( S_3 \delta(=) S_4 \)
- \( S_4 \delta(=) S_5 \)

**Dependence graph**

**Acyclic condensation**

```
S_1: DO I = 1, 10
S_3: B(I) = C(I-1)*X + Z
S_4: C(I) = 1/B(I)
x: ENDDO
S_y: DO I = 1, 10
S_2: A(I) = A(I) + B(I-1)
x: ENDDO
S_u: DO I = 1, 10
S_5: D(I) = sqrt(C(I))
v: ENDDO
```
Loop Fusion

- Combine two consecutive loops with same IV and loop bounds into one
- Fused loop must preserve all dependence relations of the original loop
- Enables more effective scalar optimizations in fused loop
- But: may reduce temporal locality

Original code has dependences
\[ S_1 \delta S_6 \text{ and } S_3 \delta S_6 \]
Fused loop has dependences
\[ S_1 \delta S_6 \text{ and } S_3 \delta(=) S_6 \text{ and } S_3 \delta(<) S_6 \]
Example

Which of the three fused loops is legal?

a)

\begin{align*}
S_1 & \text{DO I = 1, N} \\
S_2 & \quad A(I) = B(I) + 1 \\
S_3 & \quad \text{ENDDO} \\
S_4 & \text{DO I = 1, N} \\
S_5 & \quad C(I) = A(I)/2 \\
S_6 & \quad \text{ENDDO} \\
S_7 & \text{DO I = 1, N} \\
S_8 & \quad D(I) = 1/C(I+1) \\
S_9 & \quad \text{ENDDO}
\end{align*}

b)

\begin{align*}
S_x & \text{DO I = 1, N} \\
S_2 & \quad A(I) = B(I) + 1 \\
S_3 & \quad \text{ENDDO} \\
S_4 & \text{DO I = 1, N} \\
S_5 & \quad C(I) = A(I)/2 \\
S_6 & \quad \text{ENDDO} \\
S_7 & \text{DO I = 1, N} \\
S_8 & \quad D(I) = 1/C(I+1) \\
S_9 & \quad \text{ENDDO}
\end{align*}

c)

\begin{align*}
S_x & \text{DO I = 1, N} \\
S_2 & \quad A(I) = B(I) + 1 \\
S_3 & \quad \text{ENDDO} \\
S_4 & \text{DO I = 1, N} \\
S_5 & \quad C(I) = A(I)/2 \\
S_6 & \quad \text{ENDDO} \\
S_7 & \text{DO I = 1, N} \\
S_8 & \quad D(I) = 1/C(I+1) \\
S_9 & \quad \text{ENDDO}
\end{align*}
Alignment for Fusion

- Alignment for fusion changes iteration bounds of one loop to enable fusion when dependences would otherwise prevent fusion

\[ S_1 \text{ DO } I = 1, N \]
\[ S_2 \quad B(I) = T(I)/C \]
\[ S_3 \quad \text{ENDDO} \]
\[ S_4 \quad \text{DO } I = 1, N \]
\[ S_5 \quad A(I) = B(I+1) - B(I-1) \]
\[ S_6 \quad \text{ENDDO} \]

\[ S_1 \text{ DO } I = 0, N-1 \]
\[ S_2 \quad B(I+1) = T(I+1)/C \]
\[ S_3 \quad \text{ENDDO} \]
\[ S_4 \quad \text{DO } I = 1, N \]
\[ S_5 \quad A(I) = B(I+1) - B(I-1) \]
\[ S_6 \quad \text{ENDDO} \]

\[ S_x \quad B(1) = T(1)/C \]
\[ S_1 \text{ DO } I = 1, N-1 \]
\[ S_2 \quad B(I+1) = T(I+1)/C \]
\[ S_5 \quad A(I) = B(I+1) - B(I-1) \]
\[ S_6 \quad \text{ENDDO} \]
\[ S_y \quad A(N) = B(N+1) - B(N-1) \]

Loop deps:
\[ S_2 \delta(=) S_5 \]
\[ S_2 \delta(<) S_5 \]
Reversal for Fusion

- Reverse the direction of the iteration
- Only legal for loops that have no carried dependences
- Enables loop fusion by ensuring dependences are preserved between loop statements

<table>
<thead>
<tr>
<th>S1</th>
<th>DO I = 1, N</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>B(I) = T(I) * X(I)</td>
</tr>
<tr>
<td>S3</td>
<td>ENDDO</td>
</tr>
<tr>
<td>S4</td>
<td>DO I = 1, N</td>
</tr>
<tr>
<td>S5</td>
<td>A(I) = B(I+1)</td>
</tr>
<tr>
<td>S6</td>
<td>ENDDO</td>
</tr>
</tbody>
</table>

\[ \delta \]

<table>
<thead>
<tr>
<th>S1</th>
<th>DO I = N, 1, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>B(I) = T(I) * X(I)</td>
</tr>
<tr>
<td>S3</td>
<td>ENDDO</td>
</tr>
<tr>
<td>S4</td>
<td>DO I = N, 1, -1</td>
</tr>
<tr>
<td>S5</td>
<td>A(I) = B(I+1)</td>
</tr>
<tr>
<td>S6</td>
<td>ENDDO</td>
</tr>
</tbody>
</table>

\[ \delta \]

<table>
<thead>
<tr>
<th>S1</th>
<th>DO I = N, 1, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>B(I) = T(I) * X(I)</td>
</tr>
<tr>
<td>S5</td>
<td>A(I) = B(I+1)</td>
</tr>
<tr>
<td>S6</td>
<td>ENDDO</td>
</tr>
</tbody>
</table>

\[ \delta (\prec) \]

\[ S_2 \prec S_5 \]
Loop Interchange

- Compute the *direction matrix* and find which columns can be permuted without violating dependence relations in original loop nest.

\[
S_1 \text{ DO } I = 1, N \\
S_2 \text{ DO } J = 1, M \\
S_3 \text{ DO } K = 1, L \\
S_4 \text{ A}(I+1,J+1,K) = \text{ A}(I,J,K) + \text{ A}(I,J+1,K+1) \\
S_5 \text{ ENDDO} \\
S_6 \text{ ENDDO} \\
S_7 \text{ ENDDO}
\]

**Direction matrix**

\[
\begin{align*}
\langle \langle & = \rangle \\
\langle & = \rangle & \quad \langle \langle & = \rangle \\
\langle & = \rangle & \quad \langle \langle & = \rangle \\
\langle & = \rangle & \quad \langle \langle & = \rangle \\
\end{align*}
\]

Invalid

\[
\begin{align*}
\langle \langle & = \rangle \\
\langle & = \rangle & \quad \langle \langle & = \rangle \\
\langle & = \rangle & \quad \langle \langle & = \rangle \\
\langle & = \rangle & \quad \langle \langle & = \rangle \\
\end{align*}
\]

Valid
Loop Parallelization

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

```plaintext
DO I = 1, 4
  DO J = 1, 4
    S1 A(I,J+1) = A(I,J)
    ENDDO
  ENDDO
ENDDO
```

Parallelize:

```plaintext
PARALLEL DO I = 1, 4
  DO J = 1, 4
    S1 A(I,J+1) = A(I,J)
    ENDDO
  ENDDO
ENDDO
```
Vectorization (1)

- A _single-statement_ loop that carries no dependence can be vectorized

```
DO I = 1, 4
   X(I) = X(I) + C
ENDDO
```

Fortran 90 array statement

Vector operation:
```
X(1:4) + C
```

Vector operation result:
```
X(1)  X(2)  X(3)  X(4)
C     C     C     C   
```

```
X(1)  X(2)  X(3)  X(4)
```

```
X(1)  X(2)  X(3)  X(4)
```
Vectorization (2)

- This example has a loop-carried dependence, and is therefore invalid

```
S_1 \delta_{(\leq)} S_1

DO I = 1, N
S_1 X(I+1) = X(I) + C
ENDDO

S_1 X(2:N+1) = X(1:N) + C
```

Invalid Fortran 90 statement
Vectorization (3)

- Because only single loop statements can be vectorized, loops with multiple statements must be transformed using the loop distribution transformation.

- Loop has no loop-carried dependence or has forward flow dependences.

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

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

```
S1  A(2:N+1) = B(1:N) + C
S2  D(1:N) = A(1:N) + E
```
Vectorization (4)

- When a loop has backward flow dependences and no loop-independent dependences, interchange the statements to enable loop distribution.
Preliminary Transformations to Support Dependence Testing

- To determine distance and direction vectors with dependence tests require array subscripts to be in a standard form
- Preliminary transformations put more subscripts in affine form, where subscripts are linear integer functions of loop induction variables
  - Loop normalization
  - Forward substitution
  - Constant propagation
  - Induction-variable substitution
  - Scalar expansion
Forward Substitution and Constant Propagation

- Forward substitution and constant propagation help modify array subscripts for dependence analysis.

\[
\begin{align*}
N &= 2 \\
\text{DO } I &= 1, 100 \\
& \quad K = I + N \\
S_1 & \quad A(K) = A(K) + 5 \\
& \quad \text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } I &= 1, 100 \\
S_1 & \quad A(I+2) = A(I+2) + 5 \\
& \quad \text{ENDDO}
\end{align*}
\]
## Induction Variable Substitution

### Example loop with IVs

- $I = 0$
- $J = 1$
- while ($I < N$)
  - $I = I + 1$
  - $J = J + 2$
  - $K = 2 \times I$
  - $A[K] = ...$
- endwhile

### After IV substitution (IVS) (note the affine indexes)

- for $i = 0$ to $N - 1$
  - $S_1$: $... = A[2 \times i + 1]$
  - $S_2$: $A[2 \times i + 2] = ...$
- endfor

### After parallelization

- forall $(i = 0, N - 1)$
  - $... = A[2 \times i + 1]$
  - $A[2 \times i + 2] = ...$
- endforall

### GCD test to solve dependence equation

$$2^{id} - 2^{iu} = -1$$

Since 2 does not divide 1 there is no data dependence.
Example (1)

- Before and after induction-variable substitution of $KI$ in the inner loop

```plaintext
INC = 2
KI = 0
DO I = 1, 100
  DO J = 1, 100
    KI = KI + INC
    U(KI) = U(KI) + W(J)
  ENDDO
  S(I) = U(KI)
ENDDO

INC = 2
KI = 0
DO I = 1, 100
  DO J = 1, 100
    U(KI+J*INC) = U(KI+J*INC) + W(J)
  ENDDO
  KI = KI + 100 * INC
  S(I) = U(KI)
ENDDO
```
Example (2)

- Before and after induction-variable substitution of $KI$ in the outer loop

```plaintext
INC = 2
KI = 0

DO I = 1, 100
  DO J = 1, 100
    U(KI+J*INC) = U(KI+J*INC) + W(J)
  ENDDO
  KI = KI + 100*INC
S(I) = U(KI)
ENDDO

INC = 2
KI = 0

DO I = 1, 100
  DO J = 1, 100
    U(KI+(I-1)*100*INC+J*INC)
    = U(KI+(I-1)*100*INC+J*INC) + W(J)
  ENDDO
S(I) = U(KI+I*100*INC)
ENDDO
KI = KI + 100*100*INC
```
Example (3)

- Before and after constant propagation of \( \text{KI} \) and \( \text{INC} \)

```plaintext
\[
\text{INC} = 2 \\
\text{KI} = 0 \\
\text{DO} \ I = 1, 100 \\
\hspace{1cm} \text{DO} \ J = 1, 100 \\
\hspace{2cm} U(\text{KI}+(I-1)*100*\text{INC}+J*\text{INC}) \\
\hspace{3cm} = U(\text{KI}+(I-1)*100*\text{INC}+J*\text{INC}) \\
\hspace{4cm} + W(J) \\
\hspace{1cm} \text{ENDDO} \\
\hspace{1cm} S(I) = U(\text{KI}+I*100*\text{INC}) \\
\hspace{1cm} \text{ENDDO} \\
\hspace{1cm} \text{KI} = \text{KI} + 100*100*\text{INC} \\
\]
```

Affine subscripts

```plaintext
\[
\text{DO} \ I = 1, 100 \\
\hspace{1cm} \text{DO} \ J = 1, 100 \\
\hspace{2cm} U(I*200+J*2-200) \\
\hspace{3cm} = U(I*200+J*2-200) \\
\hspace{4cm} + W(J) \\
\hspace{1cm} \text{ENDDO} \\
\hspace{1cm} S(I) = U(I*200) \\
\hspace{1cm} \text{ENDDO} \\
\hspace{1cm} \text{KI} = 20000 \\
\]
```
Scalar Expansion

- Breaks anti-dependence relations by *expanding* or *promoting* a scalar into an array
- Scalar anti-dependence relations prevent certain loop transformations such as loop fission and loop interchange

```
S_1 DO I = 1, N
S_2 T = A(I) + B(I)
S_3 C(I) = T + 1/T
S_4 ENDDO

S_x IF N > 0 THEN
  S_y ALLOC Tx(1:N)
S_1 DO I = 1, N
S_2 Tx(I) = A(I) + B(I)
S_x C(I) = Tx(I) + 1/Tx(I)
S_4 ENDDO
S_z T = Tx(N)
S_u ENDIF
```

\[ S_2 \delta(=) S_3 \]
\[ S_2 \delta^{-1}(<) S_3 \]
Example

$S_1$  DO $I = 1, 10$
$S_2$  $T = A(I,1)$
$S_3$  DO $J = 2, 10$
$S_4$  $T = T + A(I,J)$
$S_5$  ENDDO
$S_6$  $B(I) = T$
$S_7$  ENDDO

$S_2$  $\delta(=) S_4$
$S_4$  $\delta(=,<=) S_4$
$S_4$  $\delta(=) S_6$
$S_2$  $\delta^{-1}(<=) S_6$

$S_1$  DO $I = 1, 10$
$S_2$  $Tx(I) = A(I,1)$
$S_3$  DO $J = 2, 10$
$S_4$  $Tx(I) = Tx(I) + A(I,J)$
$S_5$  ENDDO
$S_6$  $B(I) = Tx(I)$
$S_7$  ENDDO

$S_2$  $\delta(=) S_4$
$S_4$  $\delta(=,<=) S_4$
$S_4$  $\delta(=) S_6$

$S_2$  $\delta S_4$
$S_4$  $\delta(=,<=) S_4$
$S_4$  $\delta S_6$

$S_1$  DO $I = 1, 10$
$S_2$  $Tx(1:10) = A(1:10,1)$
$S_3$  DO $J = 2, 10$
$S_4$  $Tx(1:10) = Tx(1:10) + A(1:10,J)$
$S_5$  ENDDO
$S_6$  $B(1:10) = Tx(1:10)$

$S_2$  $\delta S_4$
$S_4$  $\delta(<=,=) S_4$
$S_4$  $\delta S_6$
Unimodular Loop Transformations

- A unimodular matrix $U$ is a matrix with integer entries and determinant $\pm 1$
- Such a matrix maps an object onto another object with exactly the same number of integer points in it
- The set of all unimodular transformations forms a group, called the *modular group*
  - The identity matrix is the identity element
  - The inverse $U^{-1}$ of $U$ exist and is also unimodular
  - Unimodular transformations are closed under multiplication: a composition of unimodular transformations is unimodular
Unimodular Loop Transformations (cont’d)

Many loop transformations can be represented as matrix operations using unimodular matrices, including loop interchange, loop reversal, loop interchange, …

\[
U_{\text{interchange}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}
\]

DO \( K = 1, 100 \)
DO \( L = 1, 50 \)
\( A(K,L) = \ldots \)
ENDDO
ENDDO

\[
U_{\text{reversal}} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}
\]

DO \( K = 1, 100 \)
DO \( L = 1, 50 \)
\( A(L,K) = \ldots \)
ENDDO
ENDDO

\[
U_{\text{skewing}} = \begin{bmatrix} 1 & 0 \\ p & 1 \end{bmatrix}
\]

DO \( K = 1, 100 \)
DO \( L = 1+p*K, 50+p*K \)
\( A(K,L-p*K) = \ldots \)
ENDDO
ENDDO
The unimodular transformation is also applied to the dependence vectors: if the result is lexicographically positive, the transformation is legal.

\[
\begin{array}{cc}
0 & 1 \\
1 & 0 \\
\end{array}
\begin{bmatrix}
1 \\
-1
\end{bmatrix} = 
\begin{bmatrix}
-1 \\
1
\end{bmatrix}
\]

Loop has distance vector \( d(i,j) = (1,-1) \) for the writes at iterations \( i=(i_1,i_2) \) and reads at iterations \( j=(j_1,j_2) \)
Unimodular Loop Transformations (cont’d)

- Transformed loop nest is given by \( A \ U^{-1} \ I \leq b \)
  - The new loop body has indices \( I' = U^{-1} I \)
  - Iteration space \( IS = \{ I \mid A \ I \leq b \} \) after transformation \( IS' = \{ I' \mid A \ U^{-1} \ I' \leq b \} \)

- Transformed loop nest needs to be normalized by means of Fourier-Motzkin elimination to ensure that loop bounds are affine expressions in more outer loop indices

\[
\begin{align*}
\text{DO } M &= 1, 3 \\
\text{DO } N &= M+1, 4 \\
A(M,N) &= \ldots \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } K &= 2, 4 \\
\text{DO } L &= 1, \text{MIN}(3,K-1) \\
A(L,K) &= \ldots \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
0 & 1 \\
1 & -1
\end{bmatrix} \leq
\begin{bmatrix}
3 \\
-1 \\
4 \\
-1
\end{bmatrix}
\]

\[
\begin{bmatrix}
M \\
N
\end{bmatrix} =
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
K \\
L
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
0 & 1 \\
1 & -1
\end{bmatrix} \leq
\begin{bmatrix}
3 \\
-1 \\
4 \\
-1
\end{bmatrix}
\]

\[
FMVE
\begin{bmatrix}
0 & 0 \\
0 & 1 \\
0 & -1
\end{bmatrix}
\begin{bmatrix}
K \\
L
\end{bmatrix} \leq
\begin{bmatrix}
2 \\
4 \\
-2
\end{bmatrix}
\]

\[
2 \leq K \leq 4
\]

\[
1 \leq L \leq 3 \text{ and } L \leq K-1
\]
Further Reading

- [Wolfe] Chapters 5, 7-9