Dependence Analysis
**Goals**

- determine what operations can be done in parallel
- determine whether the order of execution of operations can be altered

**Basic idea**

- determine a partial order on operations
- two operations not ordered in partial order
  - these operations can be done in parallel
- any new execution order of operations that is consistent with partial order is legal

**What should be the granularity of operations?**

- depends on application
  - scheduling for pipelined machines: granularity = m/c instructions
  - loop restructuring: granularity = loop iterations
Flow dependence: S1 -> S2
   (i) S1 executes before S2 in program order
   (ii) S1 writes into a location that is read by S2
Anti-dependence: S1 -> S2
   (i) S1 executes before S2
   (ii) S1 reads from a location that is overwritten later by S2
Output dependence: S1 -> S2
   (i) S1 executes before S2
   (ii) S1 and S2 write to the same location
Input dependence: S1 -> S2
   (i) S1 executes before S2
   (ii) S1 and S2 both read from the same location
Conservative Approximation:

- Real programs: imprecise information => need for safe approximation
  ‘When you are not sure whether a dependence exists, you must assume it does.’

Example:

```pascal
procedure f (X,i,j)
begin
  X(i) = 10;
  X(j) = 5;
end
```

Question: Is there an output dependence from the first assignment to the second?

Answer: If (i = j), there is a dependence; otherwise, not.

=> Unless we know from interprocedural analysis that the parameters i and j are always distinct, we must play it safe and insert the dependence.

Key notion: Aliasing: two program names may refer to the same location (like X(i) and X(j))
May-dependence vs must-dependence: More precise analysis may eliminate may-dependences
**Loop level Analysis:** granularity is a loop iteration

**Key notion:** iteration space of a loop

```
DO I = 1, 100
DO J = 1, 100
S
```

Each (I, J) value of loop indices corresponds to one point in picture.

Program order = lexicographic order on iteration space:

\[(1, 1) \leq (1, 2) \leq (1, 3) \ldots (1, 100) \leq (2, 1) \leq (2, 2) \ldots \leq (100, 100)\]

**How do we compute dependences between loop iterations?**
**Dependences in loops**

DO 10 I = 1, N
  X(f(I)) = ... 
  10 = ...X(g(I))...

- Conditions for flow dependence from iteration $I_w$ to $I_r$:
  - $1 \leq I_w \leq I_r \leq N$ *(write before read)*
  - $f(I_w) = g(I_r)$ *(same array location)*

- Conditions for anti-dependence from iteration $I_g$ to $I_o$:
  - $1 \leq I_g < I_o \leq N$ *(read before write)*
  - $f(I_o) = g(I_g)$ *(same array location)*

- Conditions for output dependence from iteration $I_{w1}$ to $I_{w2}$:
  - $1 \leq I_{w1} < I_{w2} \leq N$ *(write in program order)*
  - $f(I_{w1}) = f(I_{w2})$ *(same array location)*
Dependences in nested loops

\[
\begin{align*}
\text{DO } 10 & \ I = 1, N \\
\text{DO } 10 & \ J = 1, M \\
X(f(I,J),g(I,J)) &= \ldots \\
10 &= \ldots X(h(I,J),k(I,J))\ldots
\end{align*}
\]

Recall: \( \leq \) is the lexicographic order on iterations of nested loops.

Conditions for flow dependence from iteration \((I_1, J_1)\) to \((I_2, J_2)\):

- \((1, 1) \leq (I_1, J_1) \leq (I_2, J_2) \leq (N, M)\)
- \(f(I_1, J_1) = h(I_2, J_2)\)
- \(g(I_1, J_1) = k(I_2, J_2)\)

Anti and output dependences can be defined analogously.
In general, need more information than presence/absence of dependences.

Example

\[
\begin{align*}
&\text{DO 10 I = 1,100} & \text{DO 10 I = 1,100} \\
&\text{DO 10 J = 1,100} & \text{vs} & \text{DO 10 J = 1,100} \\
&10 \ X(I, J) = X(I-1, J+1) + 1 & 10 \ X(I, J) = X(I-1, J-1) + 1
\end{align*}
\]

Check: both loop nests have dependences.

Loop interchange:

Illegal in first loop nest (eg: (2,1) depends on (1,2))

Legal in second loop nest!

*What additional information about dependences should we compute?*
One Approach:

*Compute the full dependence relation using IP calculator*

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

Flow dependence constraints: \((I_w, J_w) \rightarrow (I_r, J_r)\)

- \((1, 1) \leq (I_w, J_w) \leq (I_r, J_r) \leq (100, 100)\)
- \(I_w = I_r - 1\)
- \(J_w = J_r + 1\)

Use IP calculator to compute the full dependence relation:
\[
\{(1, 2) \rightarrow (2, 1), (1, 3) \rightarrow (2, 2), \ldots (2, 2) \rightarrow (3, 1)\}
\]

Use full relation to check legality of loop transformations.

**Problems:**  
(i) Expensive (time/space) to compute full relation  
(ii) Expensive to look up big relation  
(iii) Variable loop bounds?
**Distance/direction:** Summarize dependence relation

Look at dependence relation from previous slide:

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

Difference between dependent iterations = \((1, -1)\). That is,

\[ (I_w, J_w) \rightarrow (I_r, J_r) \in \text{dependence relation}, \text{ implies} \]

\[ I_r - I_w = 1 \]
\[ J_r - J_w = -1 \]

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.
Summaries of dependence relations are usually *approximate*.

**Example:**

```plaintext
DO 10 I = 1,100
  10 X(2I+1) = X(I) + 1
```

Flow dependence equation: $2I_w + 1 = I_r$.
Dependence relation: \{(1 \to 3), (2 \to 5), (3 \to 7), \ldots\} (1).

No fixed distance between dependent iterations!
Since all distances are +ve, use *direction vector* instead.
Here, direction = [+]. In general, direction = [+], or [0], or [-].
Also written by some authors as (<), (=), or (>).
Intuition: [+] direction = *all* distances in range $[1, \infty)$ etc.

*Direction vectors are not exact.*
(eg): if we try to recover dependence relation from direction [+], we get bigger relation than (1):
\{(1 \to 2), (1 \to 3), \ldots, (1 \to 100), (2 \to 3), (2 \to 4), \ldots\}
Directions for Nested Loops

Assume loop nest is \((I, J)\).

If \((I_1, J_1) \rightarrow (I_2, J_2) \in \text{dependence relation}\), then

Distance = \((I_2 - I_1, J_2 - J_1)\)

Direction = \((\text{sign}(I_2 - I_1), \text{sign}(J_2 - J_1))\)

\[\begin{array}{c}
\text{Figure 1: Legal Directions for Nested Loop (I, J)}
\end{array}\]
How to compute Directions: Use IP engine

\[
\begin{align*}
&\text{DO 10 } I = 1, 100 \\
&\quad X(f(I)) = \ldots \\
&\quad 10 = \ldots X(g(I)).. 
\end{align*}
\]

Focus on flow dependences:
\[f(I_w) = g(I_r)\]
\[1 \leq I_w \leq 100\]
\[1 \leq I_r \leq 100\]

First, use inequalities shown above to test if dependence exists in any direction (called [*] direction).

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

Otherwise, determine the direction(s) of dependence.

Test for direction \(<\): add inequality \(I_w < I_r\)
Test for direction \(=\): add inequality \(I_w = I_r\)

In a single loop, direction \(>\) cannot occur.
Computing Directions: Nested Loops

Same idea as single loop: *hierarchical testing*

![Diagram of hierarchical testing for nested loops]

**Figure 2: Hierarchical Testing for Nested Loop**

**Key ideas:**

1. Refine direction vectors top down.
   
   *(eg), no dependence in \((*, *)\) direction
   
   \(\Rightarrow\) no need to do more tests.

2. Do not test for impossible directions like \((>, *)\).
In principle, we can use IP engine to compute all directions.
Reality: most subscripts and loop bounds are simple!
Engineering a dependence analyzer:

First check for simple cases.
Call IP engine for more complex cases.
Single Index Variable (SIV) subscript

DO 10 I
  DO 10 J
    DO 10 K
10   A(5,I+1,J) = ...A(N,I,K) + c

Subscripts in 1st dimension of A do not involve loop variables
⇒ subscripts called **Zero Index Variable (ZIV) subscripts**

Subscripts in 2nd dimension of A involve only one loop variable (I)
⇒ subscripts called **Single Index Variable (SIV) subscripts**

Subscripts in 3rd dimension of A involve many loop variables (J,K)
⇒ subscripts called **Multiple Index Variable (MIV) subscripts**
Separable SIV Subscript

DO 10 I
    DO 10 J
        DO 10 K
10    A(I,J,J) = ...A(I,J,K) + c

Subscripts in both the first and second dimensions are SIV.
However, index variable in first subscript \((I)\) does not appear in any other dimension

\(\Rightarrow\) separable SIV subscript

Second subscript is also SIV, but its index variable \(J\) appears in 3rd dimension as well.

\(\Rightarrow\) coupled SIV subscript
Significance of Separability: Break problem into smaller pieces

DO 10 I
   DO 10 J
      DO 10 K
10   A(I,J,J) = ...A(I,J,K) + c

Equations for flow dependence:

$I_w = I_r$
$J_w = J_r$
$J_w = K_r$

First equation can be solved separately from the other two.

If bounds on $I$ are independent of $J$ and $K$ (as here),
1st component of direction vectors can be computed independently
of 2nd and 3rd components.

In benchmarks, 80% of subscripts are separable!
Separable SIV subscript: Simple, precise tests exist.

DO 10 J
  DO 10 I
    DO 10 K
      X(aI + b, . . , .) = ..X(cI + d, . . , .) .

Equation for flow dependence: \( a * I_w + b = c * I_r + d. \)

Strong SIV subscript: \( a = c \)
\( \Rightarrow I_r - I_w = (b - d)/a \)

If \( a \) divides \( (b - d) \), and quotient is within loop bounds of \( I \), there is a dependence, and we have Ith component of the direction/distance vector.

Otherwise, no need to check other dimensions - no dependence exists!

In benchmarks, roughly 37% of subscripts are strong SIV!
Another important case:

DO 10 I
10   X(aI + b, ..., ) = ..X(cI + d, ..., )..

Weak SIV subscript: Either $a$ or $c$ is 0.

Say $c$ is 0 $\Rightarrow I_w = (d - b)/a$ and $I_r > I_w$

If $a$ divides $(d - b)$, and quotient is within loop bounds, then dependence exists with all iterations beyond $I_w$.

Important loop transformation: Index-set splitting -

It may be worth eliminating dependence by performing iterations 1..((d - b)/a) − 1 in one loop, iteration $(d - b)/a$ by itself and then the remaining iterations in another loop.
**General SIV Test Equation:**  
\[ a \cdot I_w + b = c \cdot I_r + d \]  

We can use column operations to reduce to echelon form etc.  
But usually, \( a \) and \( c \) are small integers (\( \text{mag} < 5 \)). Exploit this.

Build a table indexed by \((a, c)\) pairs for \( a \) and \( c \) between 1 and 5.

Two entries in each table position:

(i) \( \gcd(a, c) \)

(ii) one solution \((I_w, I_r) = (s, t)\) to eqn \( a \cdot I_w + c \cdot I_r = \gcd(a, c) \)

Given Equation (1), if \( a \) and \( c \) are between 1 and 5,

(i) if \( \gcd(a, c) \) does not divide \( (d - b) \), no solution

(ii) otherwise, one solution is \( (s, -t) \times (d - b) / \gcd(a, c) \)

(iii) General solution:

\[ (I_w, I_r) = n \times (c, a) / \gcd(c, a) + (s, -t) \times (d - b) / \gcd(a, c) \]

\( n \) is parameter

Case when \( a \) or \( c \) in Equation (1) are -ve: minor modification of this procedure.
**Linearization**: reduce number of equations

Not as important nowadays, but widely used earlier.

```
DO 10 I = ...
   DO 10 J = ...
       X(f(I,J),g(I,J)) = ...
   10 = ...X(h(I,J),k(I,J)).. 
```

Equations for flow dependence:

\[
\begin{align*}
    f(I_w, J_w) &= h(I_r, J_r) \\
    g(I_w, J_w) &= k(I_r, J_r)
\end{align*}
\]

We could convert system to single equation: (any constant \(C\))

\[
C \cdot f(I_w, J_w) + g(I_w, J_w) = C \cdot h(I_r, J_r) + k(I_r, J_r)
\]

Any solution of simultaneous system is soln of single equation.

But single equation may have solutions even if system does not.

\(\Rightarrow\) single equation is conservative approximation to system.
Can we choose $C$ so we do not lose precision?

If range of functions $g$ and $k$ is less than some $u$, choose $C = u$.

Easy to verify that all solutions to single equation are also solutions to system.

Common case: We know the size of array $X$ (say $\langle 1..50, 1..100 \rangle$). So $g$ and $k$ values are $< 101$. Convert

$$f(I_w, J_w) = h(I_r, J_r)$$
$$g(I_w, J_w) = k(I_r, J_r)$$

to $101 \times f(I_w, J_w) + g(I_w, J_w) = 101 \times h(I_r, J_r) + k(I_r, J_r)$

w/o loss of precision.

**Problem:** We may not know array size at compile time: eg. for array parameters, dynamically allocated arrays.
Banerjee’s test

First dependence test to take inequalities into account.

\[
\begin{align*}
\text{DO} & \quad 10 \quad I = 1, \ 100 \\
& \quad X(2I+3) = \ldots \\
& \quad 10 \quad = \ldots \ X(I+7)
\end{align*}
\]

Flow dependence constraints:

\[
\begin{align*}
1 & \leq I_w \leq I_r \leq 100 \\
2I_w + 3 & = I_r + 7
\end{align*}
\]

Inequalities: bounds on a convex region \( R \)

Equation: write as \( h(I_w, I_r) = 2I_w - I_r - 4 \).

Solving constraints is equivalent to following question:

Does \( h \) become 0 at an integer point within region \( R \)?
Banerjee’s test: checks whether \( h \) has a zero in region \( R \)

Does not check that zero is reached at integer point

**Theorem:** (Intermediate value theorem) A continuous function \( f(x, y) \) has a zero in a convex, connected region \( R \) iff

\[
\max_R f(x, y) \geq 0 \quad \text{and} \quad \min_R f(x, y) \leq 0.
\]

**Special case:** \( f \) is linear (or affine) and region \( R \) is defined by linear inequalities. In this case,

**Dantzig’s theorem:** \( f \) obtains its minimum and maximum values at the corners of region \( R \).

**Intuitive idea of Banerjee’s test:**

(i) Find corners of region \( R \)

(ii) Evaluate \( f \) at the corners and find maximum and minimum.

(iii) If \( \max \geq 0 \) and \( \min \leq 0 \), then \( f \) has a zero in \( R \).
Example:
f(x) = 3x - 2
R = [a, b] ⇒ ‘Corners’ are a and b

\[ a < b \Rightarrow 3a < 3b \Rightarrow 3a - 2 < 3b - 2 \]

So minimum value = 3a - 2; maximum value = 3b - 2

Notation: \[ x^+ := x \text{ if } x \geq 0; \text{ 0 otherwise} \]

        Similarly, \[ x^- := x \text{ if } x \leq 0; \text{ 0 otherwise} \]

Using these functions, we can compute max and min easily.

Let \( f(x) = cx + d \) and \( R = [x_{min}, x_{max}] \)

\[ \max_R f = (c)^+ x_{max} + (c)^- x_{min} + d \]
\[ \min_R f = (c)^- x_{max} + (c)^+ x_{min} + d \]

What about higher dimensions?
Function: $F(I, J)$ What are $\max_R F$ and $\min_R F$?

One way: evaluate $F$ at corners (compute $F(1, 1)$, $F(1, N)$ and $F(N, N)$) etc.

In high dimensions, hard to determine corners of region.

Important special case:

(i) variables ordered so bounds on variable $x_i$ depend only on $x_1$, $x_2, ... x_{i-1}$ (eg. $I$ bounds are constants, $J$ bounds depend only on $I$)

(ii) each variable has just one linear/affine lower and upper bound

In special case: variables can be dealt with one at a time in inner to outer order.
Example:

![Diagram](image)

Objective function: \( F(I, J) = 3I - 4J + 3 \)

Key idea: First, maximize/minimize over \( J \), keeping \( I \) fixed.

\[
\begin{align*}
\max J F(I, J) &= 3I + (-4)^+ J_{\max} + (-4)^- J_{\min} + 3 = 3I - 4 + 3 = 3I - 1 \\
\min J F(I, J) &= 3I + (-4)^- J_{\max} + (-4)^+ J_{\min} + 3 = 3 - 3I
\end{align*}
\]

Now use bounds on \( I \):

\[
\begin{align*}
\max I F(I, J) &= (3)^+ I_{\max} + (3)^- I_{\min} - 1 = 3N - 1 \\
\min I F(I, J) &= (3)^- I_{\max} + (3)^+ I_{\min} - 1 = 2
\end{align*}
\]

Summary: For special case, higher-dimensional problem is solved by repeated application of the 1-D solution.
Summary of Banerjee’s test:

Conditions:
(i) Single equality constraint
   (multiple equalities: use linearization first)
(ii) Variables can be ordered \( x_1, x_2, \ldots \) so that each upper/lower bound on \( x_i \) is an affine function of \( x_1, x_2, \ldots, x_{i-1} \)

Banerjee’s test:
(i) Interpret equality constraint as function \( F \) over domain \( R \) given by inequalities.
(ii) Maximize/minimize \( F \) over each variable separately in inner to outer order: \( x_i, x_{i-1}, \ldots, x_1 \).
(iii) If \( \max_R F \geq 0 \) and \( \min_R F \leq 0 \), then \( F \) becomes 0 at some point in \( R \).
(iv) Declare conservatively that \( F \) becomes 0 at an integer point \( \Rightarrow \) there exist integer solutions to system of inequalities/equality.
Implementation notes:

(I) Check for ZIV/separable SIV first before calling IP engine.

(II) In hierarchical testing for directions, solution to equalities should be done only once.

(III) Output of equality solver may be useful to determine distances and to eliminate some directions from consideration.

(eg) DO 10 I
    DO 10 J
        A(J) = A(J+1) + 1

Flow dependence equation: $J_w = J_r + 1 \Rightarrow distance(J) = -1$
Direction vector cannot be $(=, >)$. So only possibility is $(<, >)$: test only for this.
(IV) Array aliasing introduces complications!

procedure f(X,Y)
  DO I...
    X(I) = ...
    = ...Y(I-1)...

If X and Y may be aliased, there are may-dependences in the loop. FORTRAN convention: aliased parameters may not be modified in procedure.

(V) Pointer-based data structures: require totally different technology
(VI) Negative loop step sizes: Loop normalization

DO 10 I = 10,1,-1
    10 ....

If we use I to index into iteration space, dependence distances become -ve!

Solution: Use trip counts (0,1,...) to index loop iterations.

DO 10 I = 1,u,s
    X(I) = X(2I-5)...

Flow dependence: from trip $n_w$ to $n_r$ ⇒

\[ l + n_w \times s = 2(l + n_r \times s) - 5. \]

Distance vector = \([n_r - n_w]\)

Loop normalization: Transform all loops so low index is 0 and step size is 1. We are doing it implicitly.
(VII) Imperfectly nested loops

Distance/direction not adequate for imperfectly nested loops.

Perfectly nested loop: all statements nested within same loops

DO 10 I = 1,10
   DO 10 J = 1,10
      10 Y(I) = Y(I) + A(I,J)*X(J)

Imperfectly nested loop: triangular solve/Cholesky/LU

DO 10 I = 1,N
   DO 20 J = 1, I-1
      20 B(I) = B(I) - L(I,J)*X(J)
   10 X(I) = B(I)/L(I,I)

What should distance/direction analog be for imperfectly nested loops?
One approach: **Compute distance/direction only for common loops.**

Not adequate for many applications like imperfect loop interchange.

(row triangular solve)

```
DO 10 I = 1,N
   DO 20 J = 1, I-1
      20 B(I) = B(I) - L(I,J)*X(J)
10 X(I) = B(I)/L(I,I)
```

=>

(column triangular solve)

```
DO 10 I = 1,N
   X(I) = B(I)/L(I,I)
   DO 20 J = I+1, N
      20 B(J) = B(J) - L(I,J)*X(I)
```
What is a good dependence abstraction for imperfectly nested loops?
Conclusions

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

Recent experience:

(i) exact dependence testing is OK provided we first check for easy cases (ZIV, strong SIV, weak SIV)

(ii) IP engine is called for 3-4% of tests for direction vectors

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