Lecture 16:

Array Dependence Analysis & Parallelization

I. Data Dependence
II. Dependence Testing: Formulation
III. Dependence Testers
IV. Loop Parallelization
V. Loop Interchange

[ALSU 11.6, 11.7.8]

I. Data Dependence

```
S1: a = 1;
S2: b = a + 2;
S3: a = c - d;
      ...
S4: a = b/c;
```

Let S_i precede S_j in execution.

- Flow (true) dependence: S_i computes a data value that S_j uses.

  $\mathbf{S_i\ \delta^t\ S_j}$ E.g., $\mathbf{S_1\ \delta^t\ S_2}$ and $\mathbf{S_2\ \delta^t\ S_4}$

- Anti dependence: S_i uses a data value that S_j overwrites.

  $\mathbf{S_i\ \delta^a\ S_j}$ E.g., $\mathbf{S_2\ \delta^a\ S_3}$

- Output dependence: S_i computes a data value that S_j overwrites.

  $\mathbf{S_i\ \delta^o\ S_j}$ E.g., $\mathbf{S_1\ \delta^o\ S_3}$ and $\mathbf{S_3\ \delta^o\ S_4}$

- Input dependence: S_i uses a data value that S_j also uses.

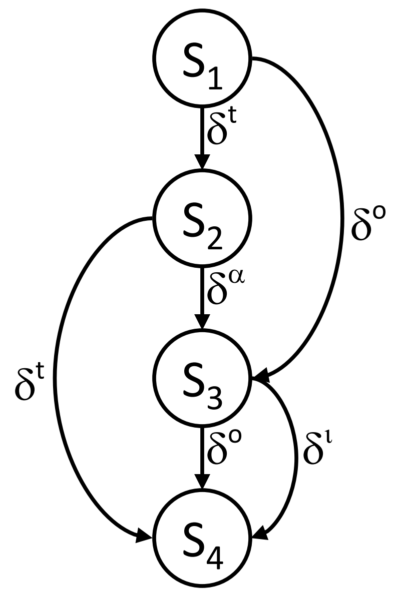
  $\mathbf{S_i\ \delta^i\ S_j}$ E.g., $\mathbf{S_3\ \delta^i\ S_4}$

  (Unlike the other 3, it is typically safe to execute S_i and S_j in parallel)

Data Dependence Graph

- Data dependence in a program may be represented using a dependence graph G=(V,E), where the nodes V represent statements in the program and the directed edges E represent dependence relations.

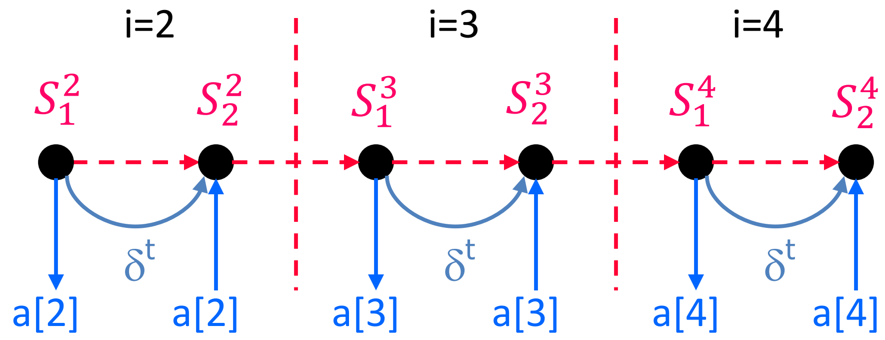
S_1: `a = 1;`
S_2: `b = a + 2;`
S_3: `a = c - d;`
`...`
S_4: `a = b/c;`



Carnegie Mellon

Array Data Dependence: Example 1

```
     for i = 2 to 4 {
S1:    a[i] = b[i] + c[i] ;
S2:    d[i] = a[i]
     }
```

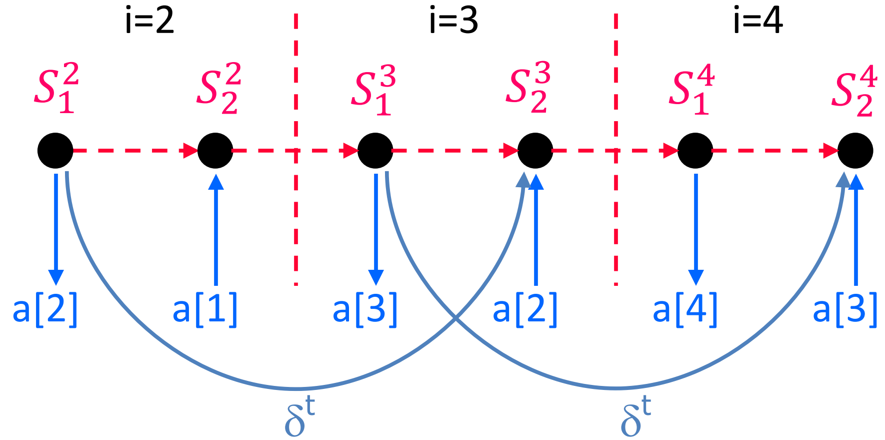


- There is an instance of S_1 that precedes an instance of S_2 in execution and S_1 produces data that S_2 uses.
- S_1 is the source of the dependence; S_2 is the sink of the dependence.
- The dependence flows between instances of statements in the same iteration (loop-independent dependence).
- The number of iterations between source and sink (dependence distance) is 0. The dependence direction is =.

$$\mathbf{S_1\ \delta^t_= \ S_2} \quad \text{or} \quad \mathbf{S_1\ \delta^t_0\ S_2}$$

Array Data Dependence: Example 2

```
      do i = 2, 4
S1:     a[i] = b[i] + c[i]
S2:     d[i] = a[i-1]
      end do
```

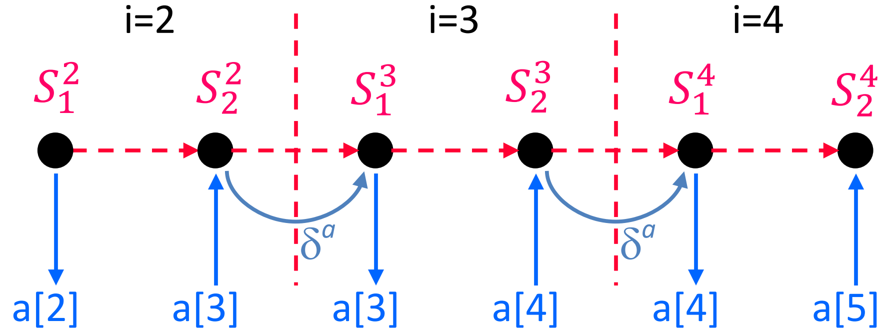


- There is an instance of S_1 that precedes an instance of S_2 in execution and S_1 produces data that S_2 uses.
- S_1 is the source of the dependence; S_2 is the sink of the dependence.
- The dependence flows between instances of statements in different iterations (loop-carried dependence).
- The dependence distance is 1. The direction is positive (<).

$\mathbf{S_1\ \delta^t_< \ S_2}$ or $\mathbf{S_1\ \delta^t_1\ S_2}$

Example 3

```
    do i = 2, 4
S1:   a[i] = b[i] + c[i]
S2:   d[i] = a[i+1]
    end do
```



- There is an instance of S_2 that precedes an instance of S_1 in execution and S_2 uses data that S_1 overwrites.
- S_2 is the source of the dependence; S_1 is the sink of the dependence.
- The dependence is loop-carried.
- The dependence distance is 1. The direction is positive (<).

$$\mathbf{S_2\ \delta^a_<\ S_1} \quad \text{or} \quad \mathbf{S_2\ \delta^a_1\ S_1}$$

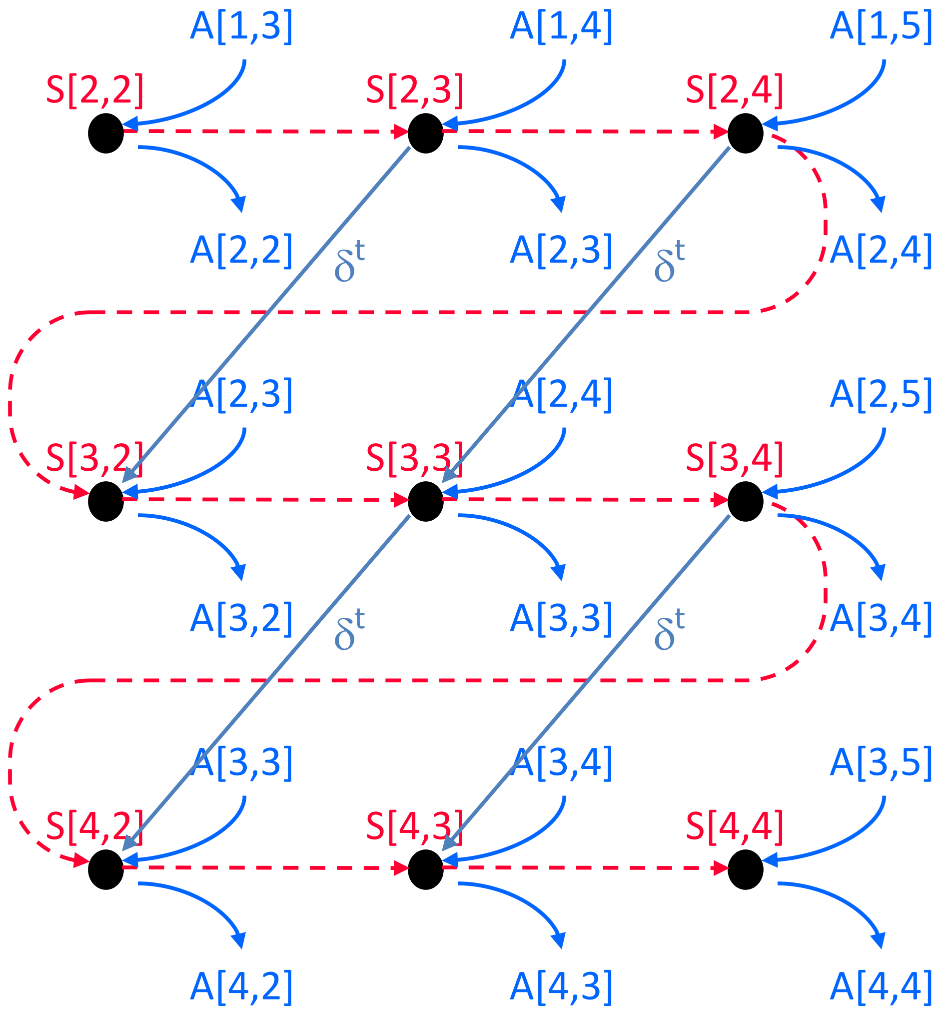
- Are you sure you know why it is $S_2\ \delta^a_<\ S_1$ even though S_1 appears before S_2 in the code?

Example 4: 2D Iteration Space

```
    do i = 2, 4
      do j = 2, 4
S:      A[i,j] = A[i-1,j+1]
      end do
    end do
```

- An instance of S precedes another instance of S and S produces data that S uses.
- S is both source and sink.
- The dependence is loop-carried.
- The dependence distance is (1,-1).

$\mathbf{S}\ \ \boldsymbol{\delta}^{\mathbf{t}}_{<,>}\ \mathbf{S}$ or $\mathbf{S}\ \boldsymbol{\delta}^{\mathbf{t}}_{\mathbf{1,-1}}\ \mathbf{S}$



II. Dependence Testing: Formulation

- Consider the following perfect nest of depth d:

"perfect" means step=1

$$
\begin{aligned}
&\text{do } I_1 = L_1, U_1\\
&\quad \text{do } I_2 = L_2, U_2\\
&\qquad \ddots\\
&\qquad\quad \text{do } I_d = L_d, U_d\\
&\qquad\qquad a(f_1(\vec{I}), f_2(\vec{I}), \cdots, f_m(\vec{I})) = \cdots\\
&\qquad\qquad \cdots = a(g_1(\vec{I}), g_2(\vec{I}), \cdots, g_m(\vec{I}))\\
&\qquad\quad \text{enddo}\\
&\qquad \cdot^{\cdot^{\cdot}}\\
&\quad \text{enddo}\\
&\text{enddo}
\end{aligned}
$$

array reference

$$a(\ \ , f_k(\vec{I}), \cdots, \ \ )$$

subscript position

subscript expression

$$\vec{I} = (I_1, I_2, \cdots, I_d)$$

$$\vec{L} = (L_1, L_2, \cdots, L_d)$$

$$\vec{U} = (U_1, U_2, \cdots, U_d)$$

$$\vec{L} \le \vec{U}$$

Affine expressions:
$c_0 + c_1 I_1 + c_2 I_2 + \cdots + c_d I_d$
for constants $c_0, c_1, \ldots, c_d$

Problem Formulation

- Dependence will exist if there exists two iteration vectors $\vec{k}$ and $\vec{j}$ such that $\vec{L} \le \vec{k} \le \vec{j} \le \vec{U}$ and:

$$
\begin{aligned}
& f_1(\vec{k}) = g_1(\vec{j}) \\
\text{and} \quad & f_2(\vec{k}) = g_2(\vec{j}) \\
\text{and} \quad & \vdots \\
\text{and} \quad & f_m(\vec{k}) = g_m(\vec{j})
\end{aligned}
$$

- That is:

$$
\begin{aligned}
& f_1(\vec{k}) - g_1(\vec{j}) = 0 \\
\text{and} \quad & f_2(\vec{k}) - g_2(\vec{j}) = 0 \\
\text{and} \quad & \vdots \\
\text{and} \quad & f_m(\vec{k}) - g_m(\vec{j}) = 0
\end{aligned}
$$

Problem Formulation - Example

```
      do i = 2, 4
S1:     a[i] = b[i] + c[i]
S2:     d[i] = a[i-1]
      end do
```

- Does there exist two iteration vectors i_1 and i_2, such that $2 \leq i_1 \leq i_2 \leq 4$ and such that $i_1 = i_2 - 1$?

- Answer: yes; $i_1=2$ & $i_2=3$ and $i_1=3$ & $i_2=4$.
- Hence, there is dependence!

- The dependence distance vector is $i_2-i_1 = 1$.
- The dependence direction vector is sign(1) = <.

Problem Formulation - Example

```
     do i = 2, 4
S1:    a[i] = b[i] + c[i]
S2:    d[i] = a[i+1]
     end do
```

- Does there exist two iteration vectors i_1 and i_2, such that $2 \leq i_1 \leq i_2 \leq 4$ and such that $i_1 = i_2 + 1$?
- Answer: yes; $i_1=3$ & $i_2=2$ and $i_1=4$ & $i_2=3$. (But, but!).
- Hence, there is dependence!
- The dependence distance vector is $i_2-i_1 = -1$.
- The dependence direction vector is sign(-1) = >.
- Is this possible? Yes: As an antidependence, not a true dependence

Carnegie Mellon

Problem Formulation - Example

```
do i = 1, 10
S1:   a[2*i] = b[i] + c[i]
S2:   d[i] = a[2*i+1]
end do
```

- Does there exist two iteration vectors i_1 and i_2, such that $1 \le i_1 \le i_2 \le 10$ and such that $2*i_1 = 2*i_2 + 1$?

- Answer: no; $2*i_1$ is even & $2*i_2+1$ is odd
- Hence, there is no dependence!

Problem Formulation

- Dependence testing is equivalent to an integer linear programming (ILP) problem of 2d variables & m+d constraints!
- An algorithm that determines if there exists two iteration vectors $\vec{k}$ and $\vec{j}$ that satisfies these constraints is called a dependence tester.

$$
\begin{aligned}
&\text{do } I_1 = L_1, U_1\\
&\quad \text{do } I_2 = L_2, U_2\\
&\qquad \ddots\\
&\qquad\quad \text{do } I_d = L_d, U_d\\
&\qquad\qquad a(f_1(\vec{I}), f_2(\vec{I}), \cdots, f_m(\vec{I})) = \cdots\\
&\qquad\qquad \cdots = a(g_1(\vec{I}), g_2(\vec{I}), \cdots, g_m(\vec{I}))\\
&\qquad\quad \text{enddo}\\
&\qquad \unicode{x22F0}\\
&\quad \text{enddo}\\
&\text{enddo}
\end{aligned}
$$

Problem Formulation

- Dependence testing is equivalent to an integer linear programming (ILP) problem of 2d variables & m+d constraints!
- An algorithm that determines if there exists two iteration vectors $\vec{k}$ and $\vec{j}$ that satisfies these constraints is called a dependence tester.
- The dependence distance vector is given by $\vec{j} - \vec{k}$.
- The dependence direction vector is give by sign($\vec{j} - \vec{k}$).
- Dependence testing is NP-complete!
- A dependence test that reports dependence only when there is dependence is said to be exact. Otherwise it is in-exact.
- A dependence test must be conservative; if the existence of dependence cannot be ascertained, dependence must be assumed.

III. Dependence Testers

- Lamport's Test.
- GCD Test.
- Banerjee's Inequalities.
- Generalized GCD Test.
- Power Test.
- I-Test.
- Omega Test.
- Delta Test.
- Stanford Test.
- etc…

Lamport's Test

- Lamport's Test is used when there is a single index variable in the subscript expressions, and when the coefficients of the index variable in both expressions are the same.

$$A(\cdots, b*i+c_1, \cdots) = \cdots$$
$$\cdots = A(\cdots, b*i+c_2, \cdots)$$

- The dependence problem: does there exist i_1 and i_2, such that $L_i \le i_1 \le i_2 \le U_i$ and such that $b*i_1 + c_1 = b*i_2 + c_2$? i.e.,

$$i_2 - i_1 = \frac{c_1 - c_2}{b}?$$

- There is integer solution if and only if $\frac{c_1 - c_2}{b}$ is integer.

- The dependence distance is $d = \frac{c_1 - c_2}{b}$ if $|d| \le U_i - L_i$
- $d > 0 \Rightarrow$ true dependence
  $d = 0 \Rightarrow$ loop independent dependence
  $d < 0 \Rightarrow$ anti dependence

Lamport's Test - Example

```
    do i = 1, n
      do j = 1, n
S:      a[i,j] = a[i-1,j+1]
      end do
    end do
```

$b*i_1 + c_1 = b*i_2 + c_2$

- $i_1 = i_2 - 1$?

  $b = 1; c_1 = 0; c_2 = -1$

  $$\frac{c_1 - c_2}{b} = 1$$

  There is dependence.

  Distance (i) is 1.

- $j_1 = j_2 + 1$?

  $b = 1; c_1 = 0; c_2 = 1$

  $$\frac{c_1 - c_2}{b} = -1$$

  There is dependence.

  Distance (j) is -1.

$\mathbf{S}\ \ \boldsymbol{\delta}^{\mathbf{t}}_{<,>}\ \mathbf{S}$ or $\mathbf{S}\ \ \boldsymbol{\delta}^{\mathbf{t}}_{\mathbf{1,-1}}\ \mathbf{S}$

Lamport's Test – Another Example

```
    do i = 1, n
      do j = 1, n
S:      a[i,2*j] = a[i-1,2*j+1]
      end do
    end do
```

$b*i_1 + c_1 = b*i_2 + c_2$

- $i_1 = i_2 - 1$?

  $b = 1; c_1 = 0; c_2 = -1$

  $$\frac{c_1 - c_2}{b} = 1$$

  There is dependence.

  Distance (i) is 1.

- $2*j_1 = 2*j_2 + 1$?

  $b = 2; c_1 = 0; c_2 = 1$

  $$\frac{c_1 - c_2}{b} = -\frac{1}{2}$$

  There is no dependence.

?

There is no dependence!

GCD Test

- Given the following equation:

  $\sum_{i=1}^{n} a_i x_i = c$ where a_i and c are integers

  an integer solution exists if and only if:

  $\gcd(a_1, a_2, \dots, a_n)$ divides c

- Problems:
  - ignores loop bounds
  - gives no information on distance or direction of dependence
  - often gcd(……) is 1 which always divides c, resulting in false dependences

GCD Test - Example

```
    do i = 1, 10
S1:   a[2*i] = b[i] + c[i]
S2:   d[i] = a[2*i-1]
    end do
```

- Does there exist two iteration vectors i_1 and i_2, such that $1 \leq i_1 \leq i_2 \leq 10$ and such that:

  $$2*i_1 = 2*i_2 - 1?$$

  or

  $$2*i_2 - 2*i_1 = 1?$$

- There will be an integer solution if and only if gcd(2,-2) divides 1.
- This is not the case, and hence, there is no dependence!

GCD Test: Another Example

```
      do i = 1, 10
S1:     a[i] = b[i] + c[i]
S2:     d[i] = a[i-100]
      end do
```

- Does there exist two iteration vectors i_1 and i_2, such that $1 \leq i_1, i_2 \leq 10$ and such that:

  $i_1 = i_2 - 100$?

  or

  $i_2 - i_1 = 100$?

- There will be an integer solution if and only if gcd(1,-1) divides 100.
- This is the case, and hence, there is dependence! Or is there?

No: check loop bounds. Shows a limitation of GCD.

Dependence Testing: Complications

- Unknown loop bounds:

```
      do i = 1, N
S1:     a[i] = a[i+10]
      end do
```

What is the relationship between N and 10?

- Triangular loops:

```
      do i = 1, N
        do j = 1, i-1
S:        a[i,j] = a[j,i]
        end do
      end do
```

Must impose $j < i$ as an additional constraint.

More Complications

- User variables:

```
      do i = 1, 10
S1:     a[i] = a[i+k]
      end do
```

Same problem as unknown loop bounds, but occur due to some loop transformations (e.g., loop bounds normalization).

```
      do i = L, H
S1:     a[i] = a[i-1]
      end do
```

⇓

```
      do i = 1, H-L
S1:     a[i+L] = a[i+L-1]
      end do
```

More Complications: Scalars

```
      do i = 1, N
S1:     x = a[i]
S2:     b[i] = x
      end do
```

⇒

```
      do i = 1, N
S1:     x[i] = a[i]
S2:     b[i] = x[i]
      end do
```

privatization

```
      j = N-1
      do i = 1, N
S1:     a[i] = a[j]
S2:     j = j - 1
      end do
```

⇒

```
      do i = 1, N
S1:     a[i] = a[N-i]

      end do
```

IV. Loop Parallelization

- A dependence is said to be carried by a loop if the loop is the outermost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is loop-independent.

```
do i = 2, n-1
   do j = 2, m-1
      a(i, j)     = ...
       ...        = a(i, j)

      b(i, j)     = ...
       ...        = b(i, j-1)

      c(i, j)     = ...
       ...        = c(i-1, j)
   end do
end do
```

Loop Parallelization

- A dependence is said to be carried by a loop if the loop is the outermost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is loop-independent.

Outermost loop with a non "=" direction carries dependence!

```
do i = 2, n-1
   do j = 2, m-1
      a(i, j)     = ...
       ...        = a(i, j)

      b(i, j)     = ...
       ...        = b(i, j-1)

      c(i, j)     = ...
       ...        = c(i-1, j)
   end do
end do
```

$\delta^t_{=,=}$ — loop-independent

$\delta^t_{=,<}$ — carried by loop j

$\delta^t_{<,=}$ — carried by loop i

Loop Parallelization

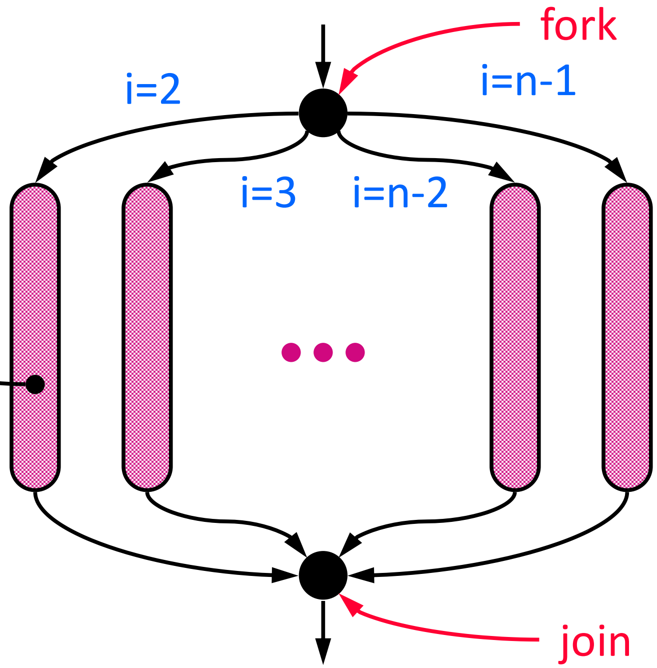
The iterations of a loop may be executed in parallel with one another if and only if no dependences are carried by the loop!

Carnegie Mellon

Loop Parallelization - Example

$\delta^{t}_{=,<}$

```
do i = 2, n-1
    do j = 2, m-1
        b(i, j)      = …
        …            = b(i, j-1)
    end do
end do
```

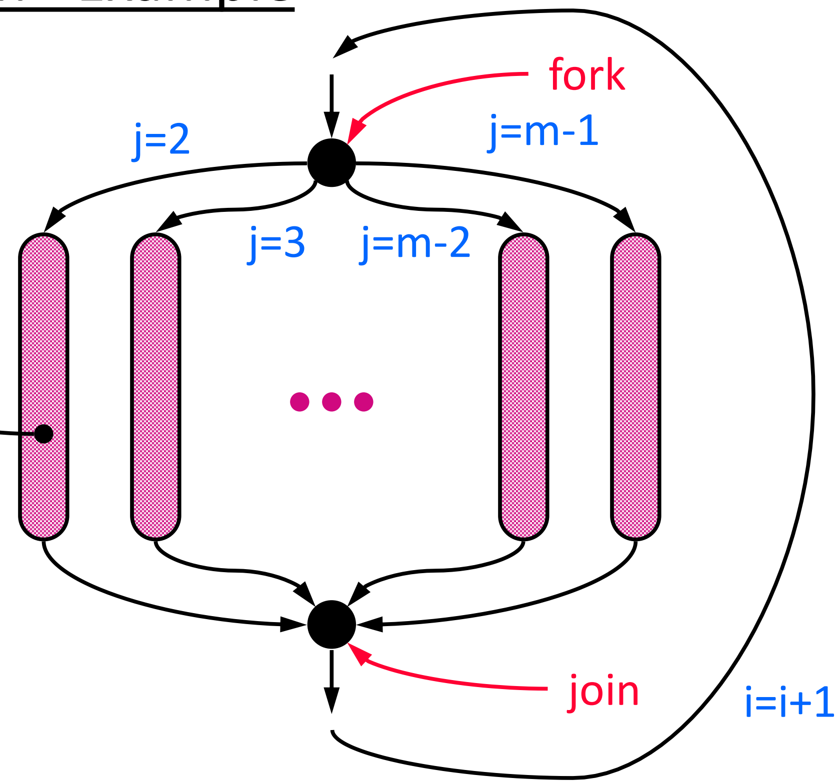


- Iterations of loop j must be executed sequentially, but the iterations of loop i may be executed in parallel!
- Outer loop parallelism

Loop Parallelization - Example

$\delta^t_{<,=}$

```
do i = 2, n-1
   do j = 2, m-1
      b(i, j)     = ...
      ...         = b(i-1, j)
   end do
end do
```

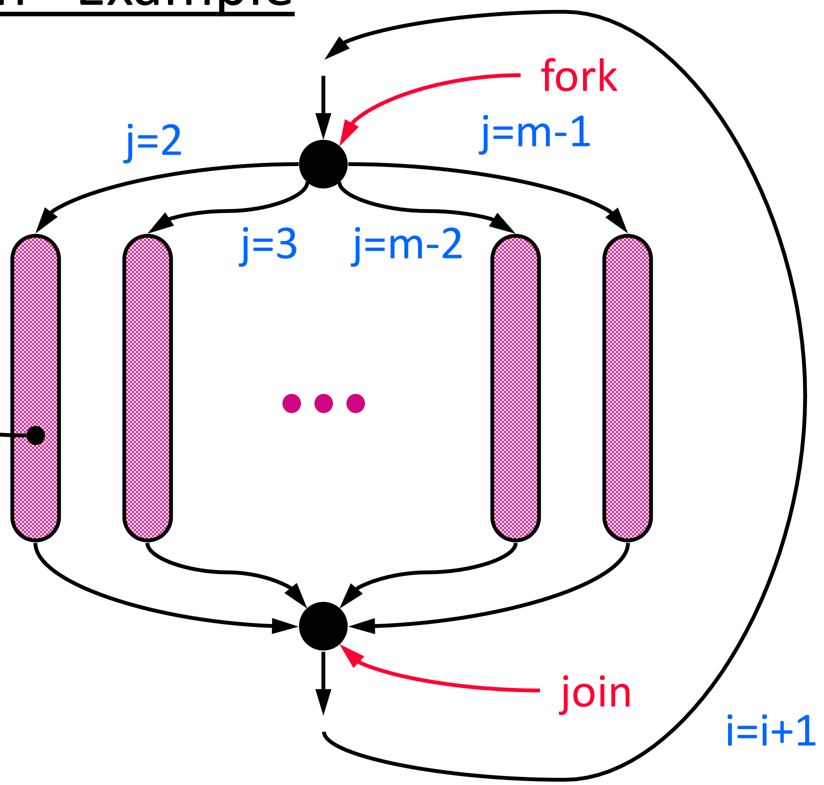


- Iterations of loop i must be executed sequentially, but the iterations of loop j may be executed in parallel!
- Inner loop parallelism (Vectorization, SIMD)

Loop Parallelization - Example

$\delta^{t}_{<,<}$

```
do i = 2, n-1
   do j = 2, m-1
      b(i, j)     = ...
        ...       = b(i-1, j-1)
   end do
end do
```



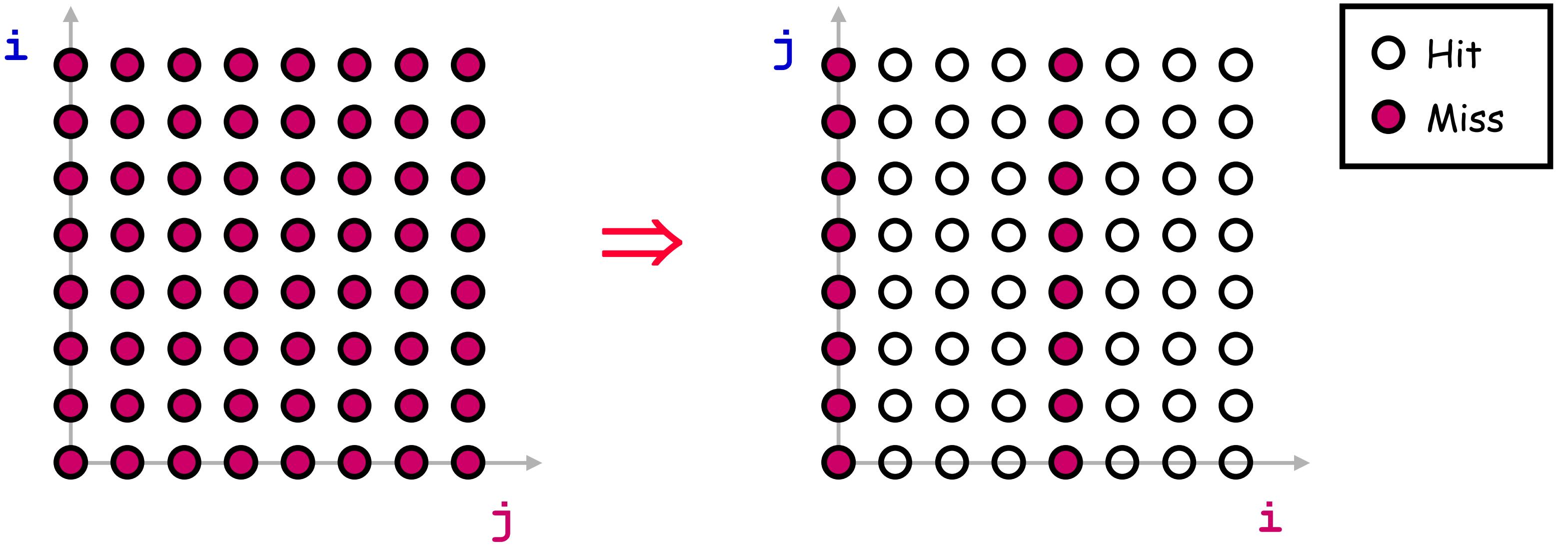
- Iterations of loop i must be executed sequentially, but the iterations of loop j may be executed in parallel! Why?
- Inner loop parallelism

V. Loop Interchange

Recall: Used to improve spatial locality

```
for i = 0 to N-1
   for j = 0 to N-1
      A[j][i] = i*j;
```

```
for j = 0 to N-1
   for i = 0 to N-1
      A[j][i] = i*j;
```



Assume row major order, N large, 4 elements per cache line

Loop Interchange

Can also be used to improve the granularity of parallelism!

```
do i = 1, n
  do j = 1, n
     a[i,j] = b[i,j]
     c[i,j] = a[i-1,j]
  end do
end do
```

$\Rightarrow$

```
do j = 1, n
  do i = 1, n
     a[i,j] = b[i,j]
     c[i,j] = a[i-1,j]
  end do
end do
```

$\delta^t_{<,=}$

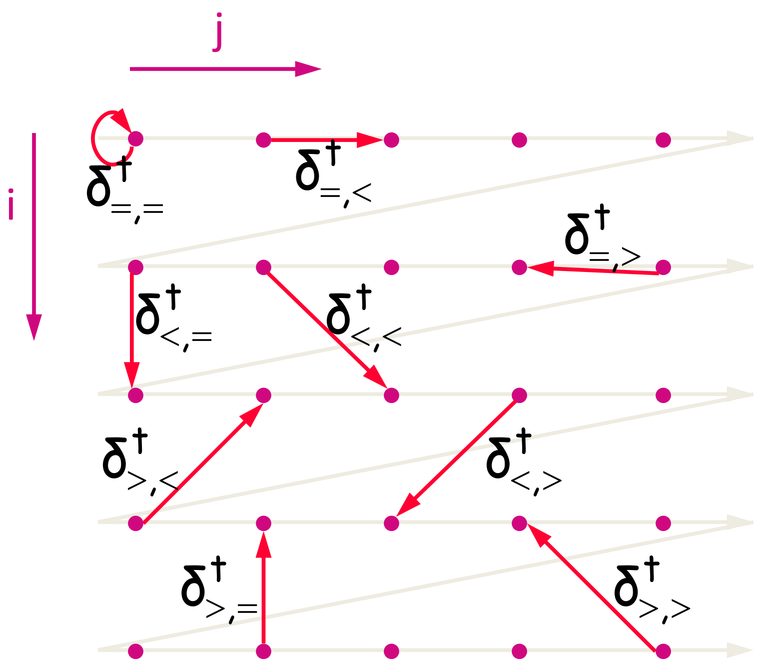
$\delta^t_{=,<}$

Inner loop parallelism

Outer loop parallelism

When Is Loop Interchange Legal?

```
do i = 1,n
  do j = 1,n
     … a[i,j] …
  end do
end do
```

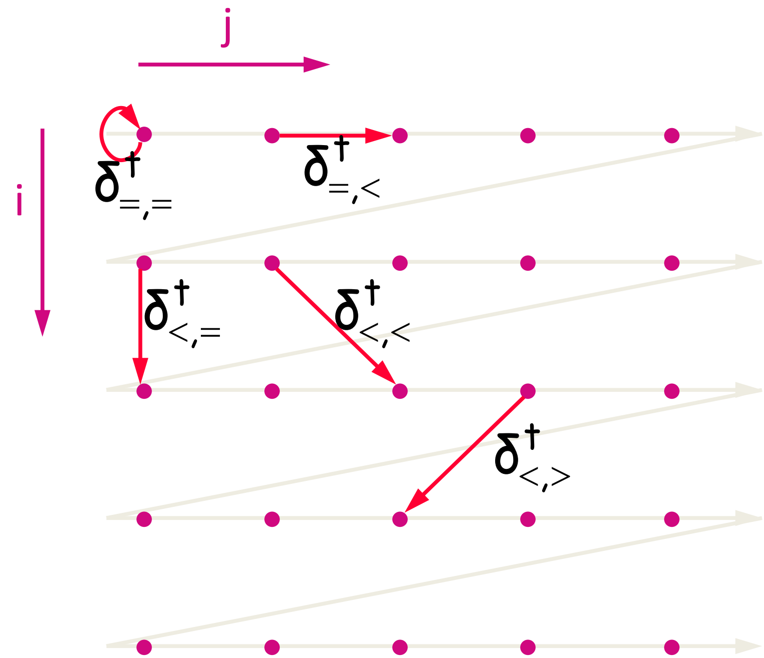



```
do j = 1,n
  do i = 1,n
     … a[i,j] …
  end do
end do
```

When Is Loop Interchange Legal?

Focus only on true dependences (i.e., lexicographically positive dependences)

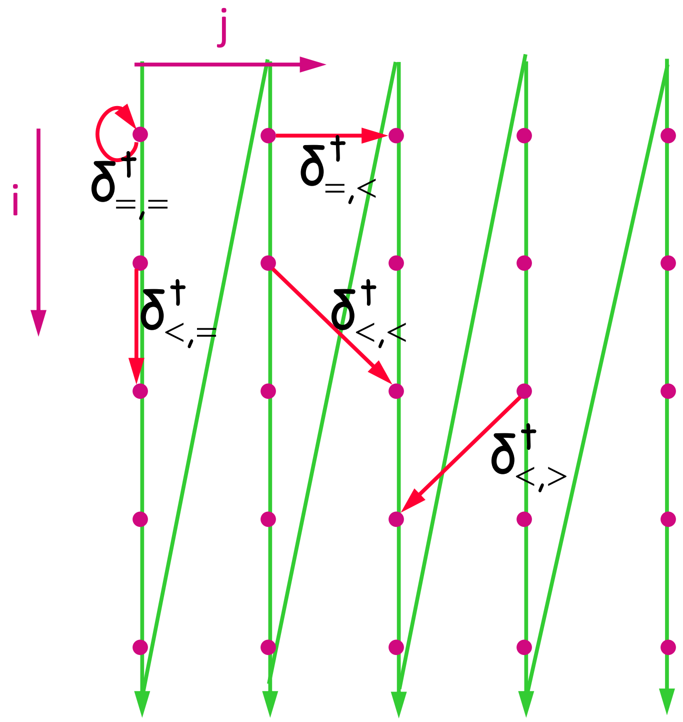
```
do i = 1,n
  do j = 1,n
     … a[i,j] …
  end do
end do
```




```
do j = 1,n
  do i = 1,n
     … a[i,j] …
  end do
end do
```

When Is Loop Interchange Legal?

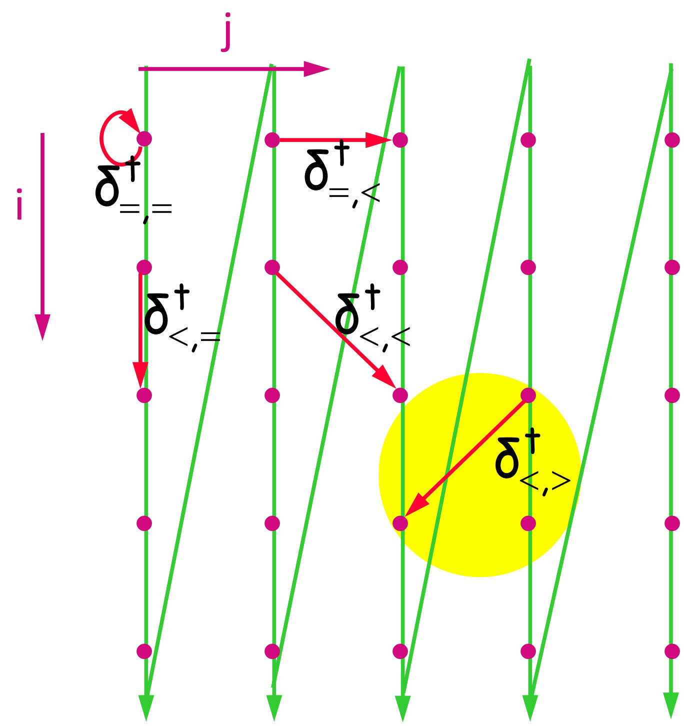
```
do i = 1,n
  do j = 1,n
     … a[i,j] …
  end do
end do
```




```
do j = 1,n
  do i = 1,n
     … a[i,j] …
  end do
end do
```

When Is Loop Interchange Legal?

```
do i = 1,n
  do j = 1,n
     … a[i,j] …
  end do
end do
```




```
do j = 1,n
  do i = 1,n
     … a[i,j] …
  end do
end do
```

When is loop interchange legal?

when the “interchanged” dependences remain lexiographically positive!

Today's Class: Array Dependence Analysis & Parallelization

I. Data Dependence
II. Dependence Testing: Formulation
III. Dependence Testers
IV. Loop Parallelization
V. Loop Interchange

Wednesday's Class

- Guest Lecture:
  Chris Fallin on Data-Structure Aware Distinctness Analysis