CS 314 Principles of Programming Languages

Lecture 19: Parallelism and Dependence Analysis

Prof. Zheng Zhang

Rutgers University

November 14, 2018

Review: Dependence Definition

Bernstein's Condition: — There is a data dependence from statement (instance) S_1 to statement S_2 (instance) if

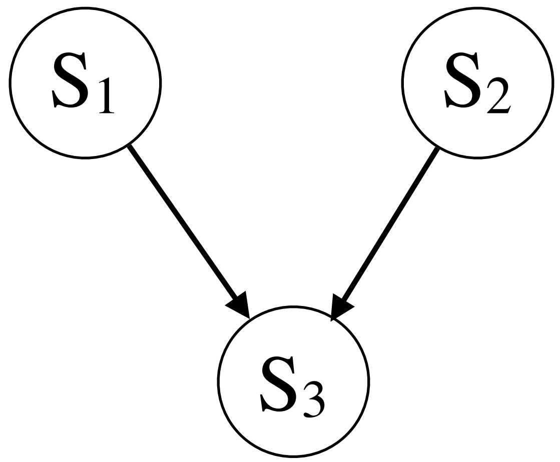
- Both statements (instances) access the same memory location(s)
- One of them is a write
- There is a run-time execution path from S_1 to S_2

Example:

S_1: pi = 3.14

S_2: R = 5

S_3: Area = pi * R^2



Data Dependence Classifications

"S_2 depends on S_1" — ($S_1\ \delta\ S_2$)

True (flow) dependence

occurs when S1 writes a memory location that S2 later reads (RAW).

Anti dependence

occurs when S1 reads a memory location that S2 later writes (WAR).

Output dependence

occurs when S1 writes a memory location that S2 later writes (WAW).

Input dependence

occurs when S1 reads a memory location that S2 later reads (RAR).

Review: Dependence Testing

Single Induction Variable (SIV) Test

- Single loop nest with constant lower (LB) and upper (UB) bound, and step 1.

```
for i = LB, UB, 1
      ...
endfor
```

- Two array references as affine function of loop induction variable

```
for i = LB, UB, 1
    R1: X(a*i + c1) = ...
    R2:        ... = X(a*i + c2)
endfor
```

Question: Is there a true dependence between R1 and R2?

Review: Dependence Testing

```
for i = LB, UB, 1
    R1: X(a*i + c1) = ...
    R2:       ... = X(a*i + c2)
endfor
```

There is a dependence between R1 and R2 **iff**

$$\exists\, i, i':\ LB \leq i \leq i' \leq UB \text{ and } (a*i+c_1) = (a*i'+c_2)$$

where **i** and **i′** represent two iterations in the iteration space. This means that in both iterations, the same element of array X is accessed.

So let's just solve the equation:

$$(a * i + c_1) = (a * i' + c_2) \quad \Rightarrow \quad (c_1 - c_2)/a = i' - i = \Delta d$$

There is a dependence iff

- Δd is an integer value
- $UB - LB \geq \Delta d \geq 0$

Simple Dependence Testing

- **Examples:**

```
for (i = 1; i <= 100; i++) {
  S1:  A[i] = ...
  S2:  ...= A[i - 1]
}
```

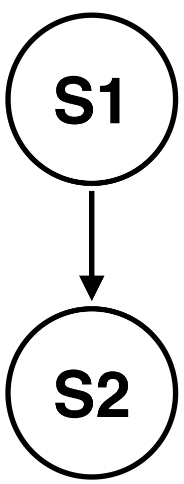
```
float Z[100];
for (i =0; i < 12; i++) {
    S: Z[ i+10 ] = Z[i];
}
```

1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?

Simple Dependence Testing

- **Examples:**

```
for (i = 1; i <= 100; i++) {
  S1:  A[i] = ...
  S2:  ...= A[i - 1]
}
```

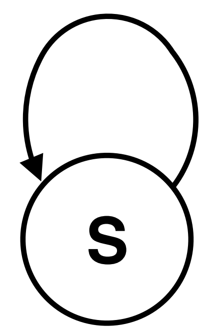


True Dependence
(read after write):
Wt: A[i] in S1 →
Rd: A[i'-1] in S2

i' = i +1

$\Delta d = 1$

```
float Z[100];
for (i =0; i < 12; i++) {
   S: Z[ i+10 ] = Z[i];
}
```



True Dependence
(read after write):
Wt: Z[i+10] in S →
Rd: Z[i'] in S

i' = i +10

$\Delta d = 10$

Simple Dependence Testing

- **More Examples:**

```
for (i = 1; i <= 100; i++) {
    R1:  X(i) = ...
    R2:  ... = X(i + 2)
}
```

```
for (i = 3; i <= 15, i++) {
    S1:  X(2 * i) = ...
    S2:  ... = X(2 * i - 1)
}
```

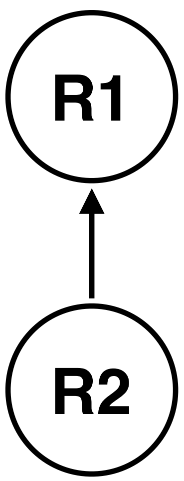
1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?

Simple Dependence Testing

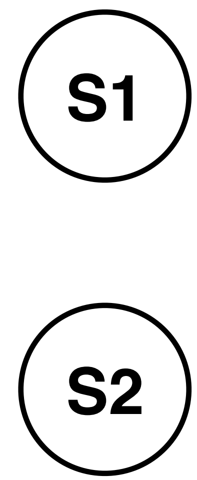
- **More Examples:**

```
for (i = 1; i <= 100; i++) {
    R1:  X[i] = ...
    R2:  ... = X[i + 2]
}
```

```
for (i = 3; i <= 15, i++) {
    S1:  X[2 * i] = ...
    S2:  ... = X[2 * i - 1]
}
```



Anti Dependence (write after read): Rd: X[i+2] in R2 → Wt: X[i’] in R1



No dependence!

Review: Automatic Parallelization

We will use **loop analysis** as an example to describe automatic dependence analysis and parallelization.

Assumptions:

1. We only have scalar and subscripted variables (no pointers and no control dependence) for loop dependence analysis.
2. We focus on ***affine loops***: both loop bounds and memory references are affine functions of loop induction variables.

A function $f(x_1, x_2, ..., x_n)$ is **affine** if it is in such a form:

$\mathbf{f} = c_0 + c_1 * \boldsymbol{x_1} + c_2 * \boldsymbol{x_2} + \ldots + c_n * \boldsymbol{x_n}$, where c_i are all constants

Review: Affine Loops

Three spaces

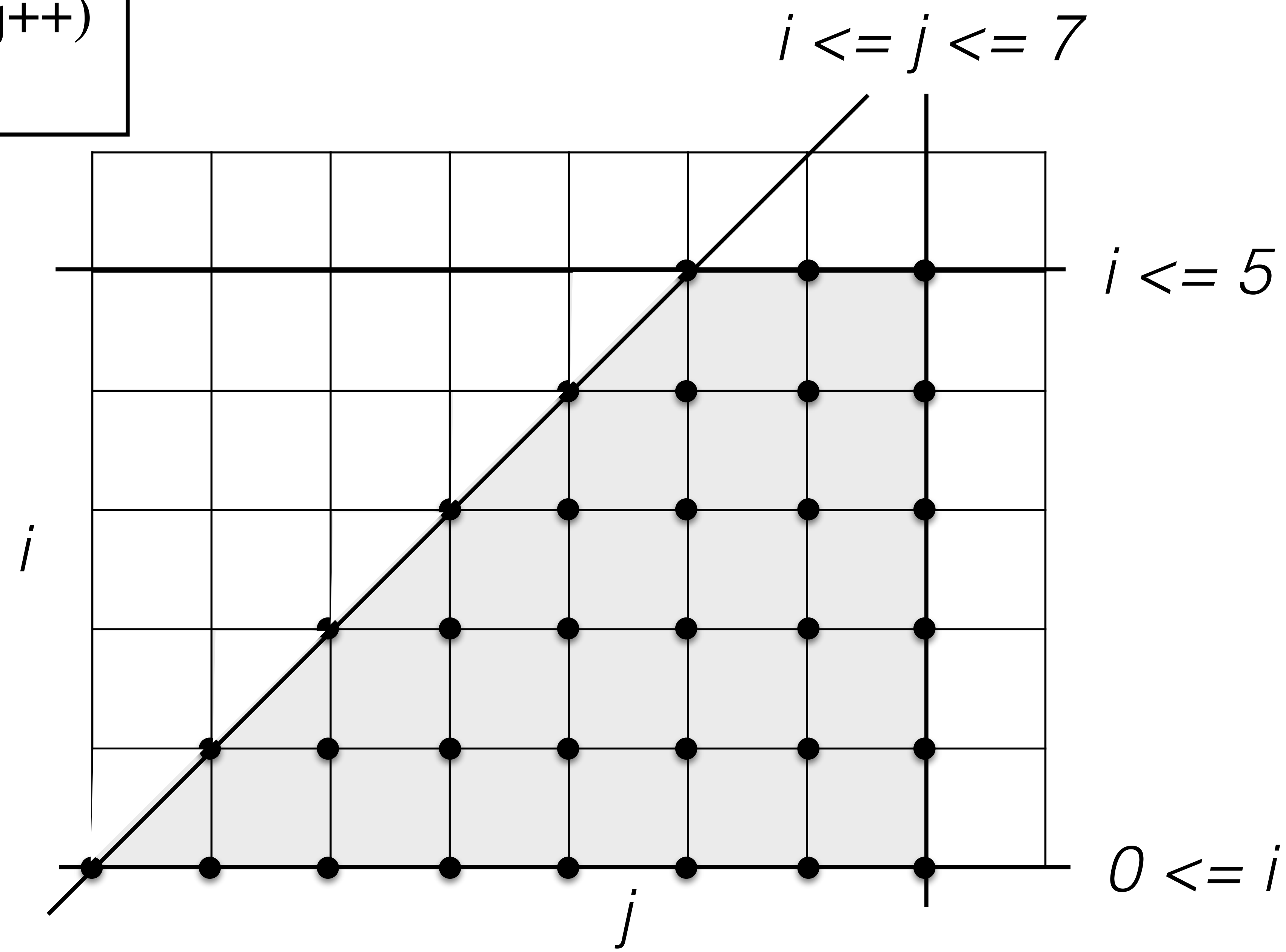
- Iteration space
  - The set of dynamic execution instances
  - i.e. the set of value vectors taken by loop indices
  - A *k*-dimensional space for a *k*-level loop nest
- Data space
  - The set of array elements accessed
  - An *n*-dimensional space for an *n*-dimensional array
- Processor space
  - The set of processors in the system
  - In analysis, we may pretend there are unbounded # of virtual processors

Iteration Space

- **Example**

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

$0 <= i <= 5$
$i <= j <= 7$



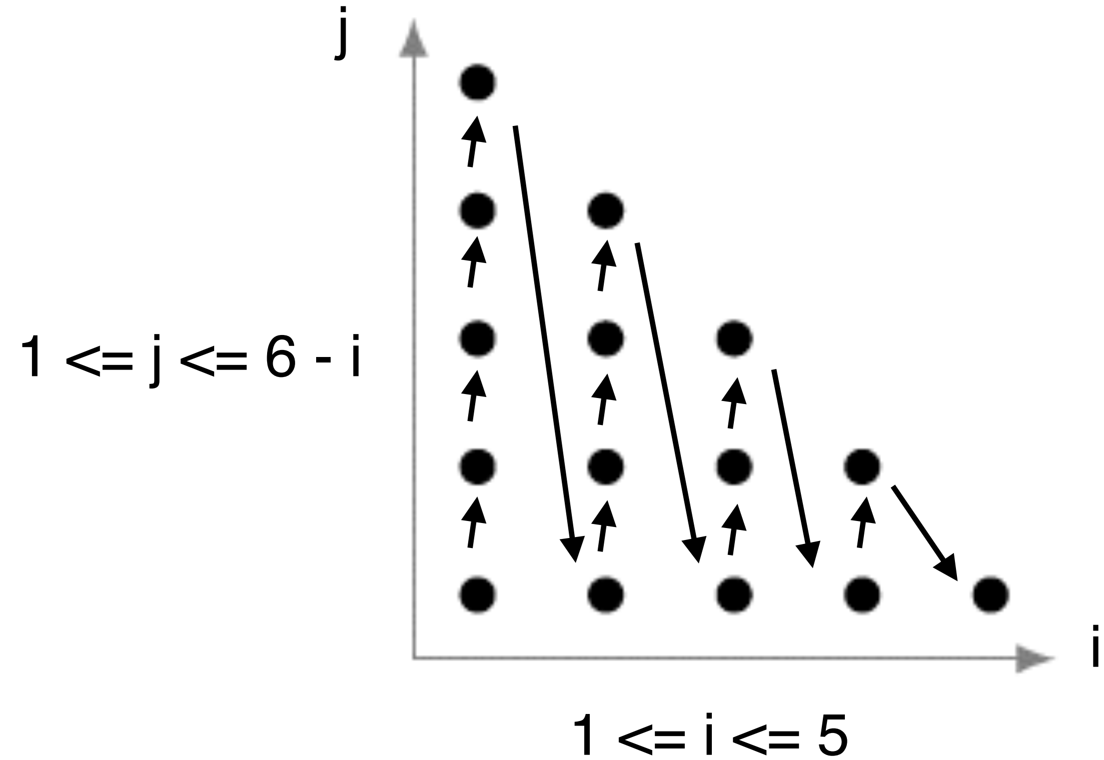
Lexicographical Order

- Order of sequential loop executions
- Sweeping through the space in an ascending lexicographic order:

  (i, j) <= (i', j') iff one of the two conditions is satisfied

  1. i <= i'
  2. i = i' & j <= j'

```
for (i = 1; i <= 5; i++)
   for (j = 1; j <= 6 - i; j++)
      Z[j, i] = 0;
```



Dependence Test

Given

$$
\begin{array}{l}
\text{do } i_1 = L_1, U_1 \\
\quad \ldots \\
\quad\quad \text{do } i_n = L_n, U_n \\
\quad\quad\quad S1: \quad A[\, f_1(\, i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)\,] = \ldots \\
\quad\quad\quad S2: \quad \ldots \;=\; A[\, g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)\,]
\end{array}
$$

A dependence between statement (instance) S_1 and S_2, denoted $S_1\ \delta\ S_2$, indicates that the S_1 instance, the source, must be executed before S_2 instance, the sink on some iteration of the loop nest.

Let α & β be a vector of n integers within the ranges of the lower and upper bounds of the n loops.

Does $\exists\ \alpha, \beta$ in the loop iteration space, s.t.

$$f_k(\alpha) = g_k(\beta) \qquad \forall k, 1 \le k \le m?$$

Dependence Test

Given

$$
\begin{aligned}
&\text{do } i_1 = L_1, U_1 \\
&\quad \ldots \\
&\qquad \text{do } i_n = L_n, U_n \\
&\qquad\quad S1: \quad A[\, f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)\,] = \ldots \\
&\qquad\quad S2: \quad \ldots = A[\, g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)\,]
\end{aligned}
$$

Example: consider the two memory references X[i, j] and X[i, j-1]

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```

For X[i,j]: $f_1(i,j) = i$,
$f_2(i,j) = j$;

For X[i,j-1]: $g_1(i,j) = i$,
$g_2(i,j) = j - 1$;

Dependence Test as Integer Linear Programming Problem

Does $\exists\ \alpha, \beta$ in the loop iteration space, s.t.

$$f_k(\alpha) = g_k(\beta) \qquad \forall k, 1 \le k \le m?$$

```
for (i=1; i<=100; i++)
   for (j=1; j<=100; j++){
      S1: X[i,j] = X[i,j] + Y[i-1, j];
      S2: Y[i,j] = Y[i,j] + X[i, j-1];
   }
```

α: (i_1, j_1)

β: (i_2, j_2)

Consider the two memory references:

S1(α): **$X[i_1, j_1]$**, S2(β): **$X[i_2, j_2-1]$**

Do such (i_1, j_1), (i_2, j_2) exist?

If there is dependence, then

$i_1 = i_2$

$j_1 = j_2 - 1$

And

(i_1, j_1): $1 <= i_1 <= 100$, $1 <= j_1 <= 100$,

(i_2, j_2): $1 <= i_2 <= 100$, $1 <= j_2 <= 100$,

Dependence Test as Integer Linear Programming Problem

Does $\exists$ α, β in the loop iteration space, s.t.

$$f_k(\alpha) = g_k(\beta) \qquad \forall k, 1 \le k \le m?$$

```
for (i=1; i<=100; i++)
   for (j=1; j<=100; j++){
      S1: X[i,j] = X[i,j] + Y[i-1, j];
      S2: Y[i,j] = Y[i,j] + X[i, j-1];
   }
```

α: (i_1, j_1)

β: (i_2, j_2)

Consider the two memory references:

S1(α): **X[i_1, j_1]**, S2(β): **X[i_2, j_2-1]**

access the same memory location →

$i_1 = i_2$

$j_1 = j_2 - 1$

loop bounds constraint →

$1 <= i_1 <= 100$

$1 <= j_1 <= 100$

$1 <= i_2 <= 100$

$1 <= j_2 <= 100$

Do such (i_1, j_1), (i_2, j_2) exist?

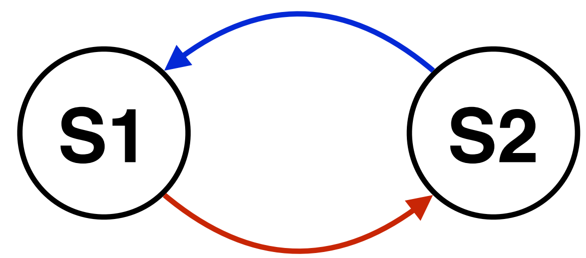
Does there exist a solution to this integer linear programming (ILP) problem?

Back to this Example

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i, j] = X[i,j] + Y[i-1, j];
        S2: Y[i, j] = Y[i,j] + X[i, j-1];
    }
```

Dependence in the "i" loop

True Dependence
(RAW)
Wt: Y[i, j] in S2
→ Rd: Y[i'-1, j'] in S1



True Dependence
(RAW)
Wt: X[i, j] in S1 →
Rd: X[i', j'-1] in S2

Dependence in the "j" loop

Access the same memory location →

$i_1 = i_2$

$j_1 = j_2 - 1$

Loop bounds constraints →

$1 <= i_1 <= 100$

$1 <= j_1 <= 100$

$1 <= i_2 <= 100$

$1 <= j_2 <= 100$

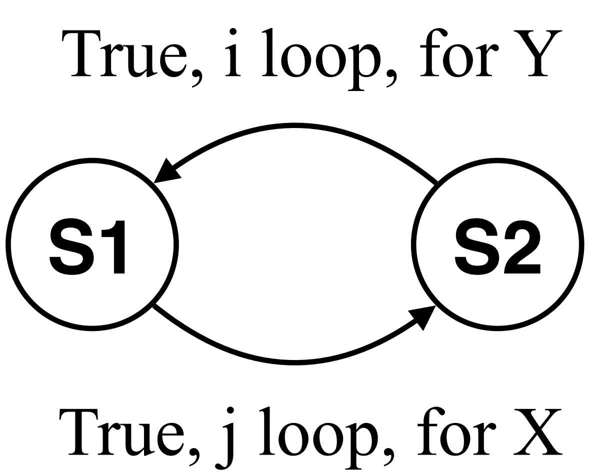
(Only showing the ILP problem for the dependence marked in red.)

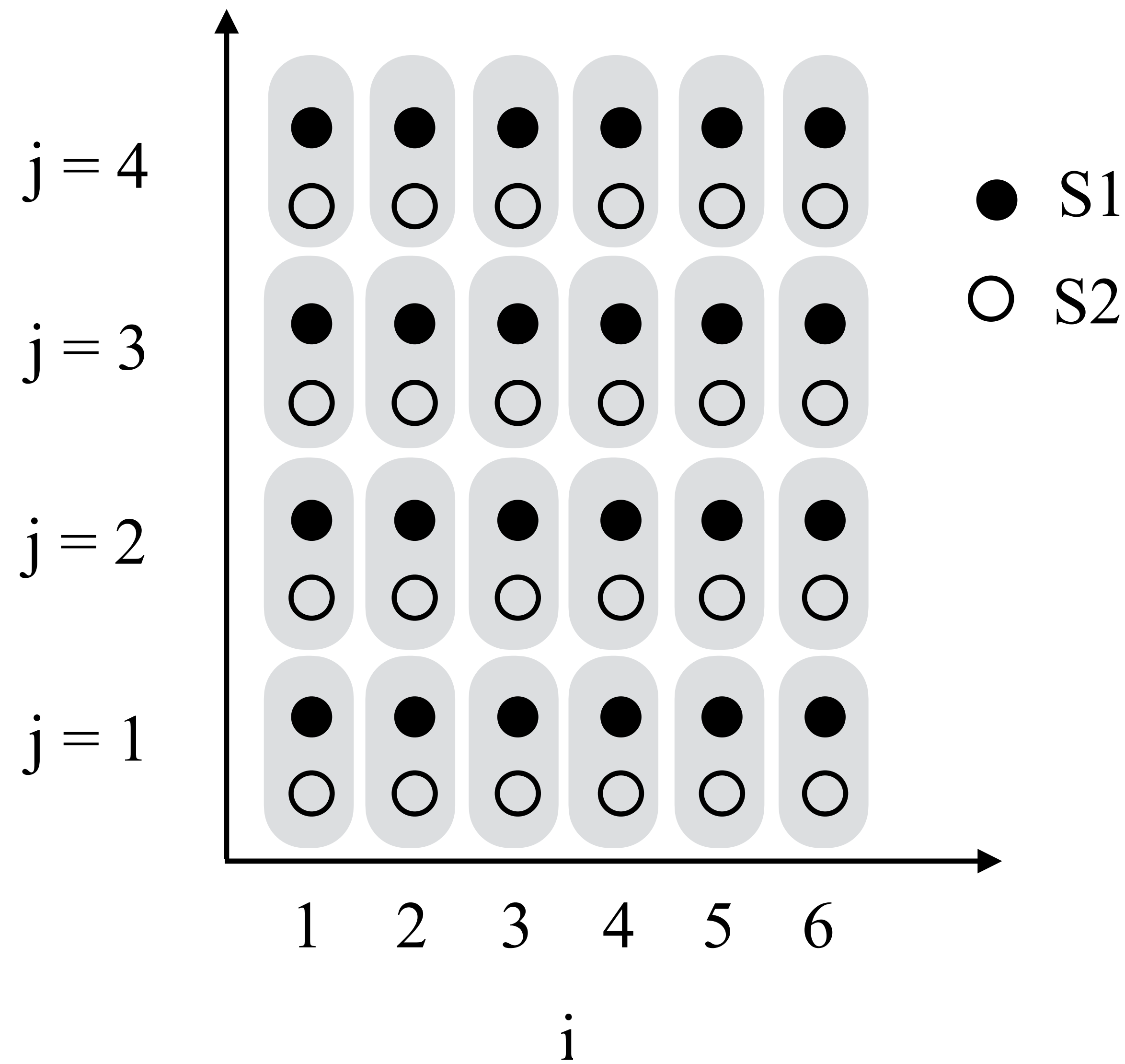
Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

Dependence from **S2**(1,1) to **S1**(2,1)

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```



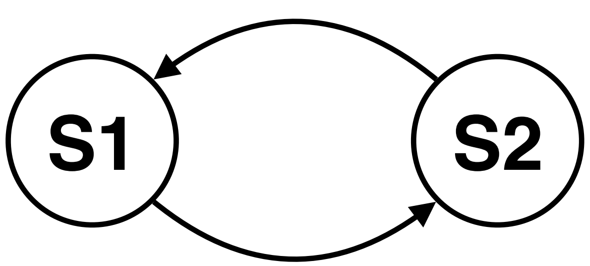


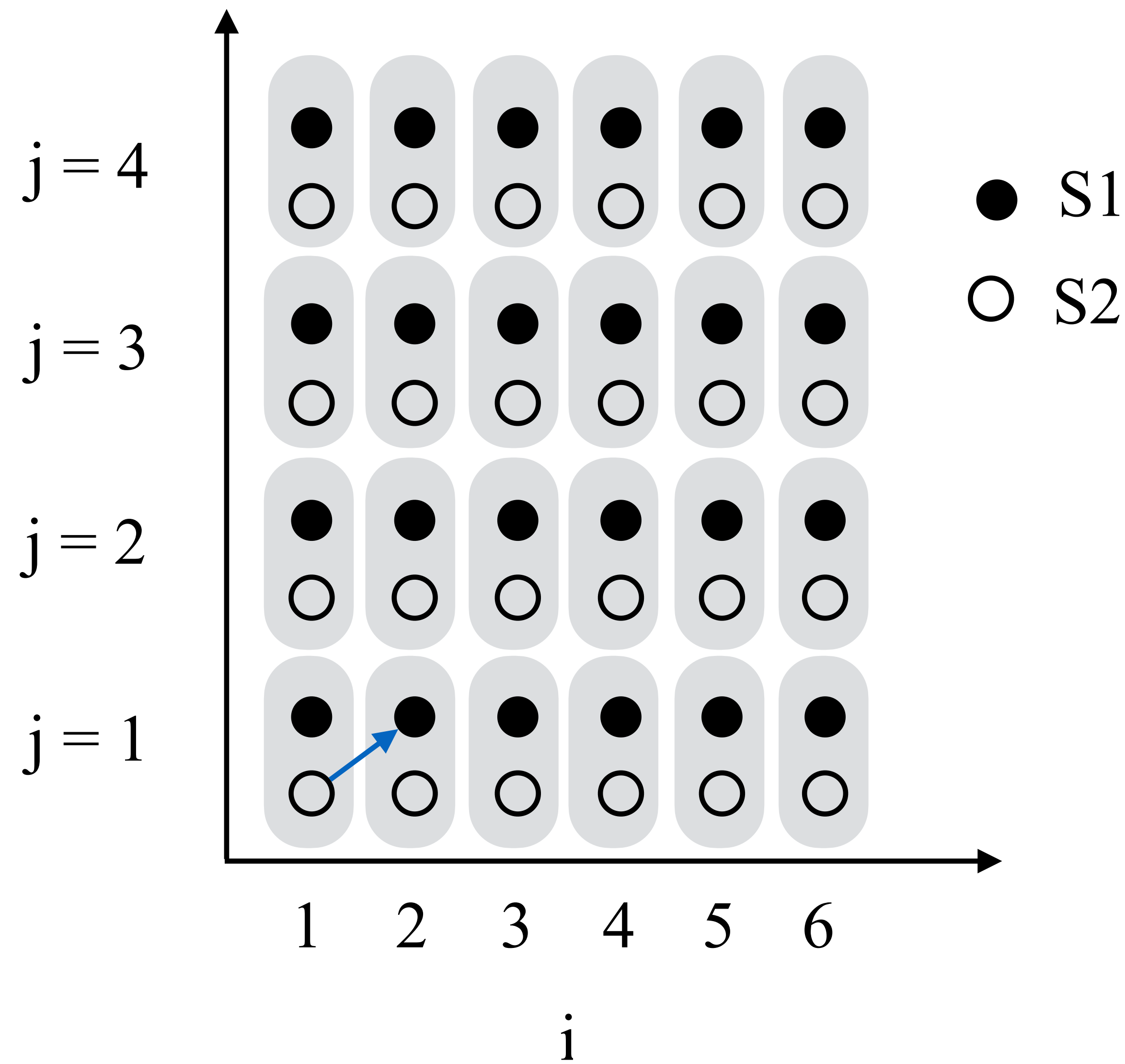
Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

Dependence from **S2**(1,1) to **S1**(2,1)
for Y[,] memory reference

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```



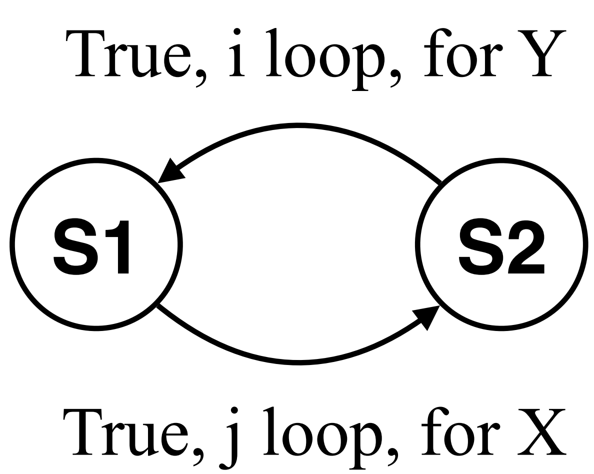


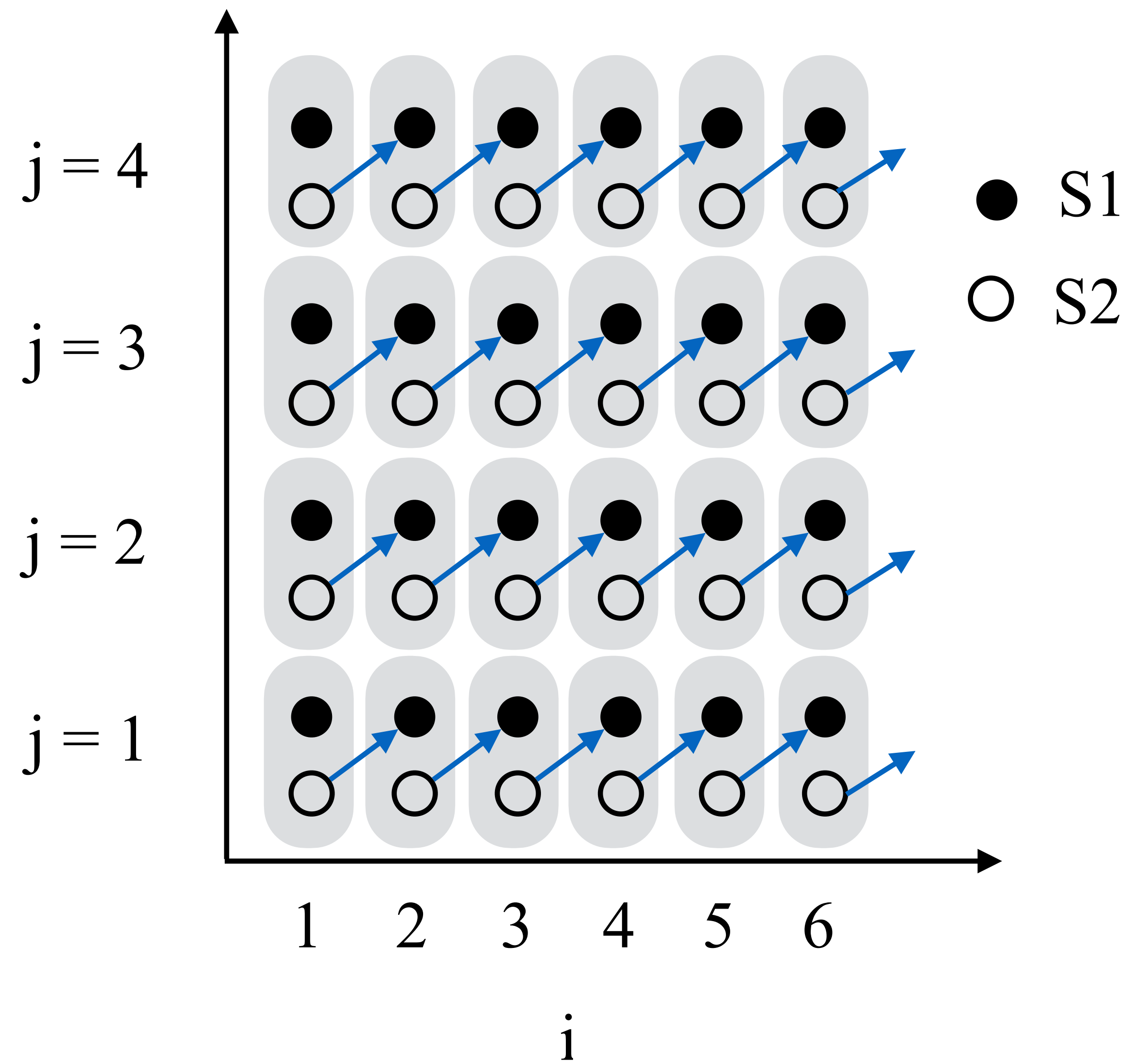
Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

Dependence from **S2**(1,1) to **S1**(2,1)
for Y[,] memory reference

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```





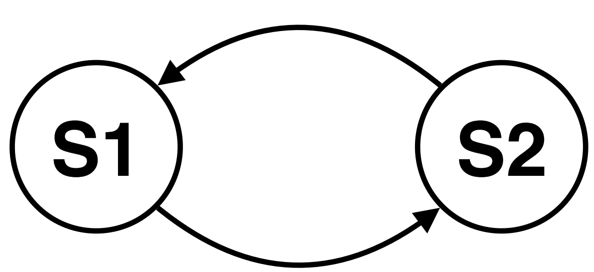
Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

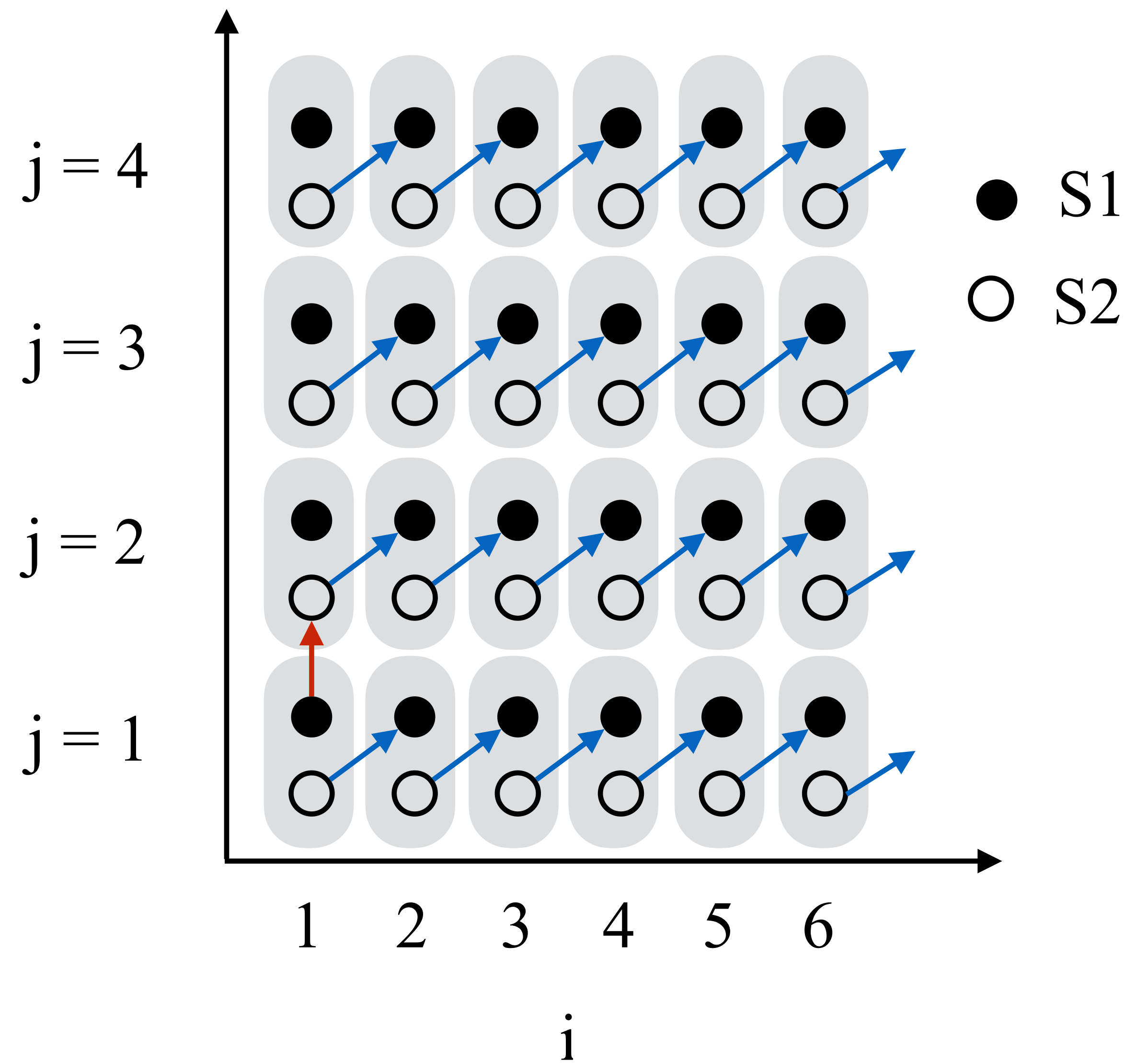
Dependence from **S1**(1,1) to **S2**(1,2)
for X[,] memory reference

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```

True, i loop, for Y



True, j loop, for X

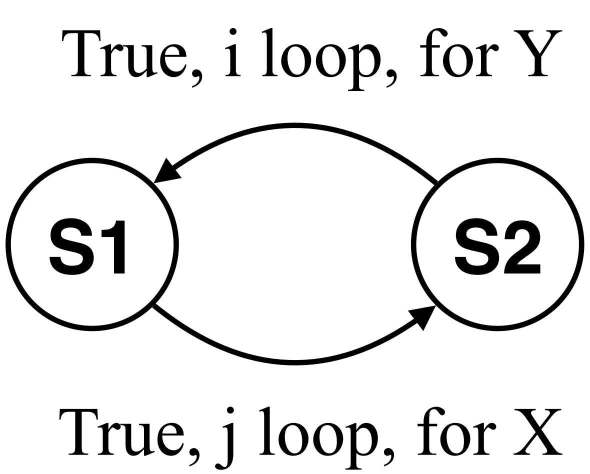


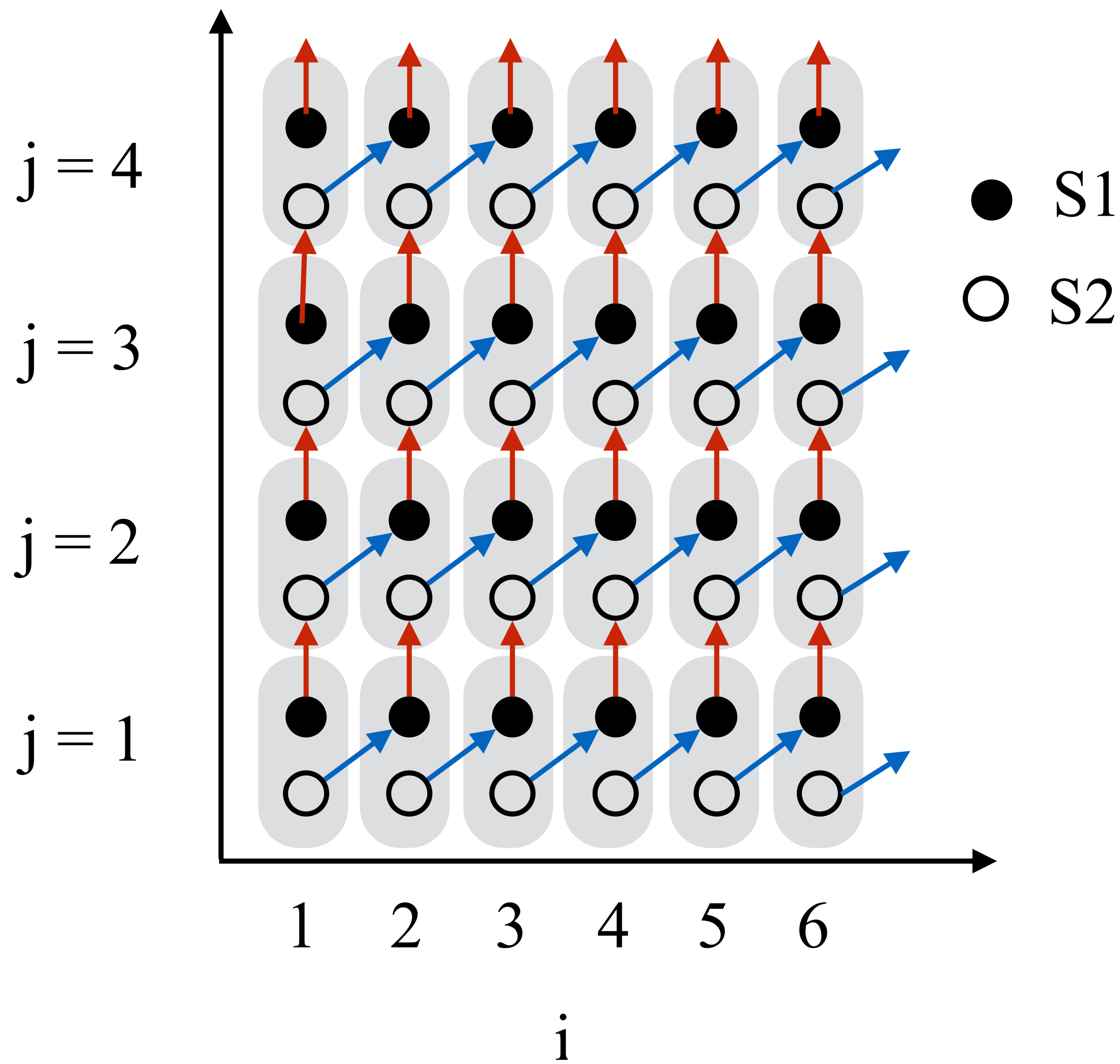
Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

Dependence from **S1**(1,1) to **S2**(1,2)
for X[,] memory reference

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```





Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

Dependence from **S1**(1,1) to **S1**(1,2)

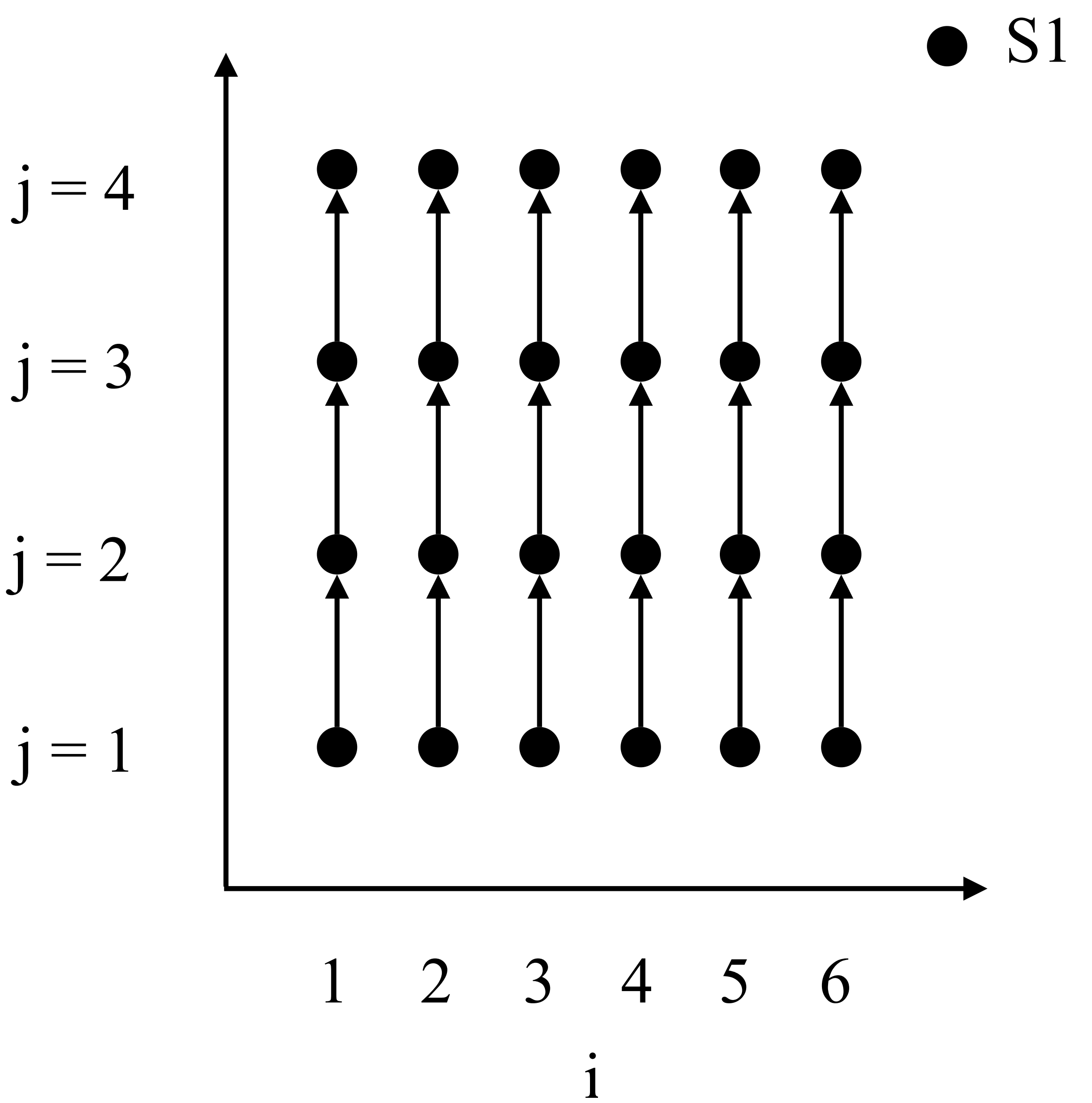
```
do i = 1, N
    do j = 1, N
      S1: A[i, j] = A[i, j - 1]
```

Write in $S_1(1,1)$ to Read in $S_1(1,2)$

↓

Write: $S_1(i, j)$ to Read in $S_1(i, j+1)$

Which loop can be parallelized?
The "i" loop or the "j" loop?



Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

Dependence from **S1**(1,1) to **S1**(1,2)

```
doall i = 1, N
    do j = 1, N
      S1: A[i, j] = A[i, j - 1]
```

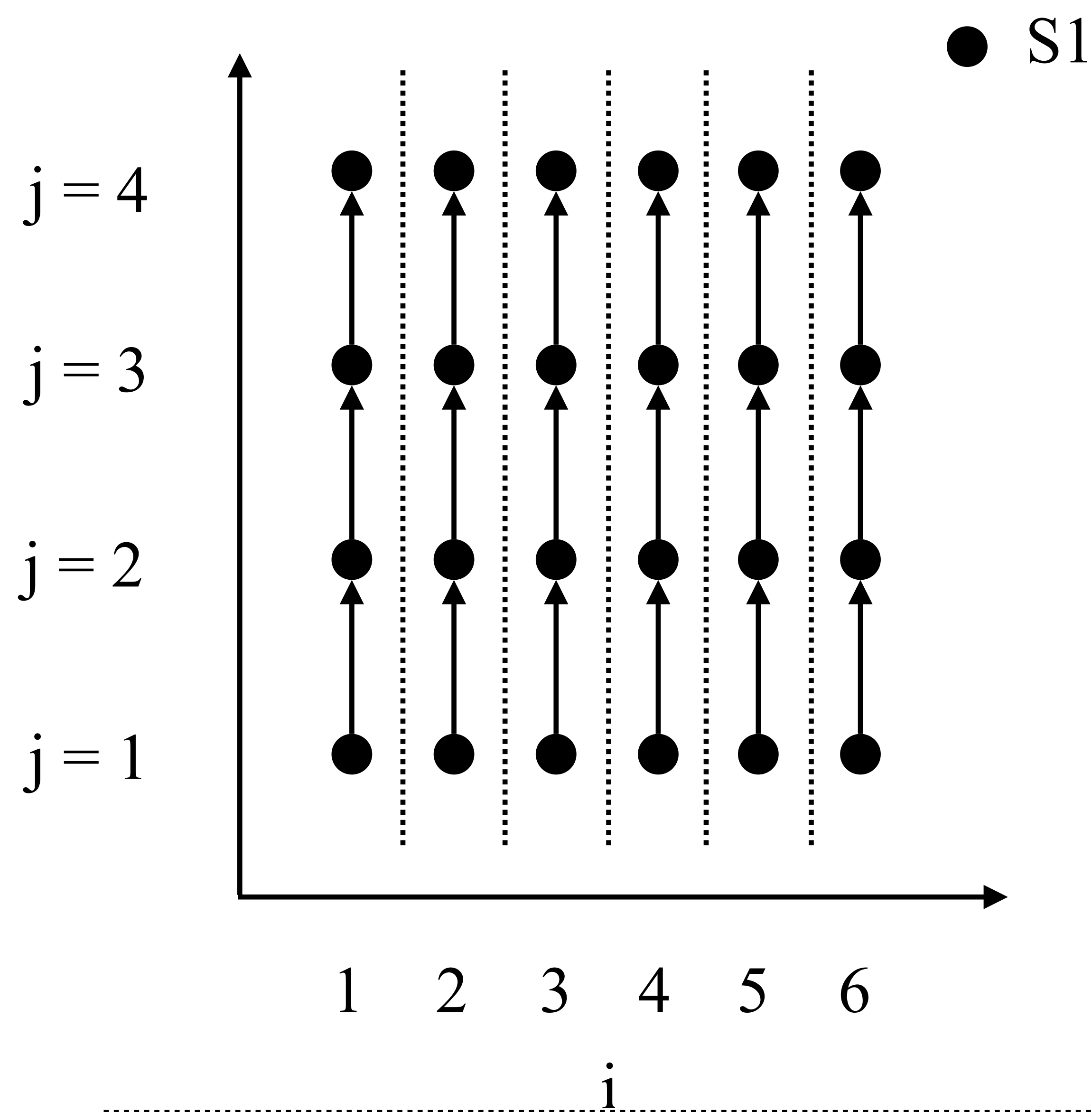
Write in $S_1(1,1)$ to Read in $S_1(1,2)$

↓

Write: $S_1(i, j)$ to Read in $S_1(i, j+1)$

Which loop can be parallelized?
The "i" loop or the "j" loop?

Answer: the "i" loop



doall loop means all iterations in the loop can run in parallel

Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially

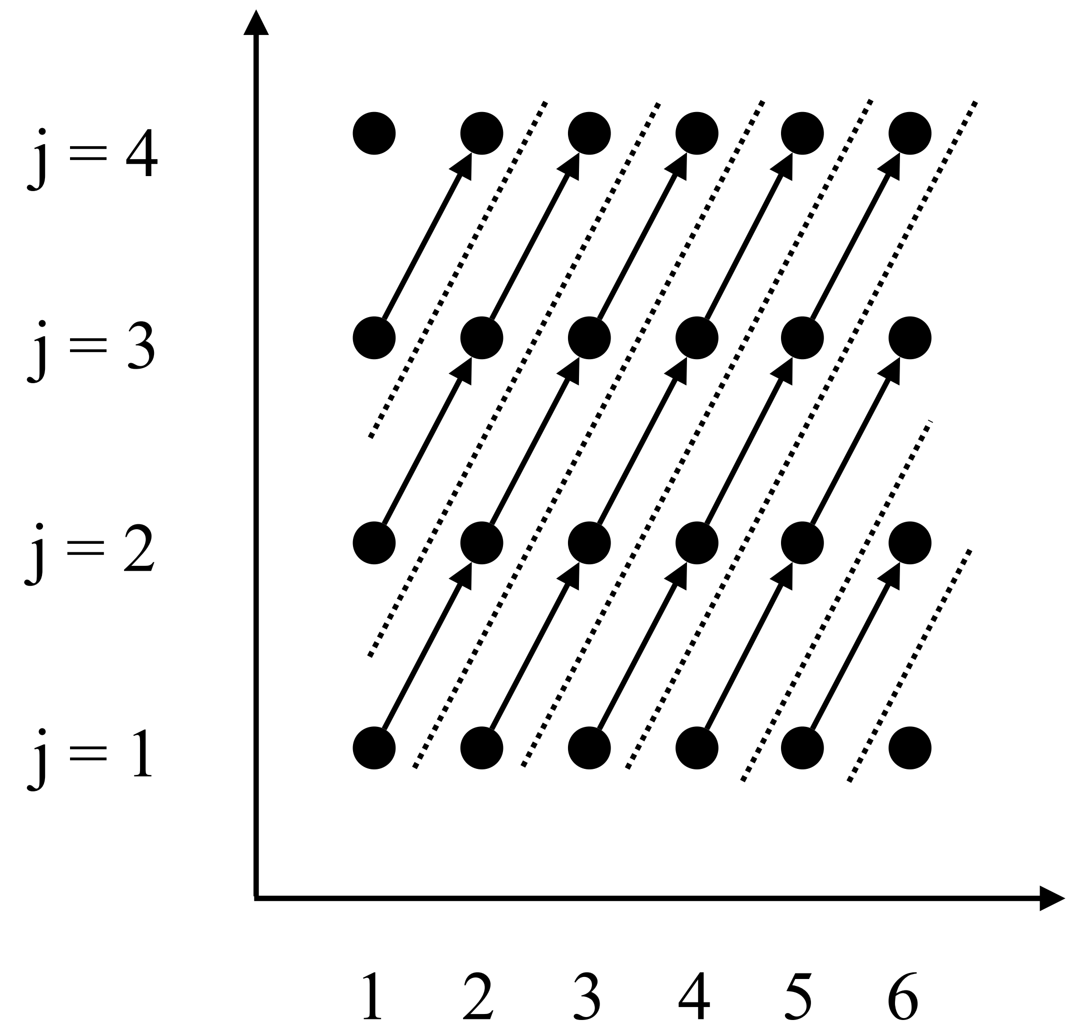
Dependence from **S1**(1, 1) to **S1**(2, 2)

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

Write in $S_1(1,1)$ to Read in $S_1(2,2)$

↓

Write in $S_1(i, j)$ to Read $S_1(i+1, j+1)$



Can either the "i" loop or the "j" loop be parallelized? (assuming no synchronization is allowed)

The hyperplane is j - i = "a constant"

Dependence and Parallelization

- Dependence in affine loops modeled as a hyperplane
- Iterations along the same hyperplane must execute sequentially
- **Iterations on different hyperplanes can execute in parallel**

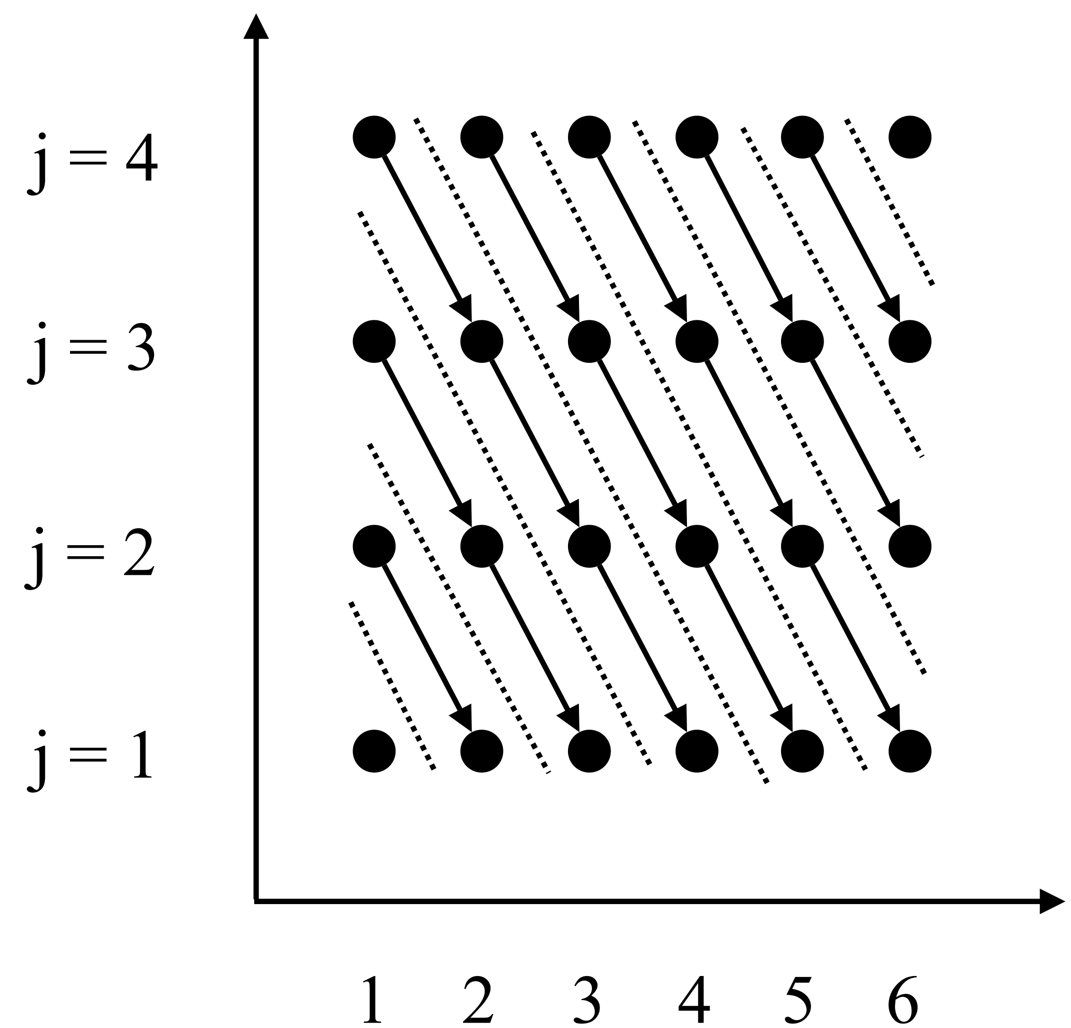
Dependence from **S1**(1, 2) to **S1**(2, 1)

```
do I = 1, N
    do J = 1, N
    S1: A[I, J] = A[I-1, J+1]
```

Write in $S_1(1,2)$ to Read in $S_1(2,1)$

↓

Write in $S_1(i, j)$ to Read in $S_1(i-1,j+1)$



The hyperplane is j + i = "a constant"

Distance Vector

The number of iterations between two accesses to the same memory location, usually represented as a **distance vector**.

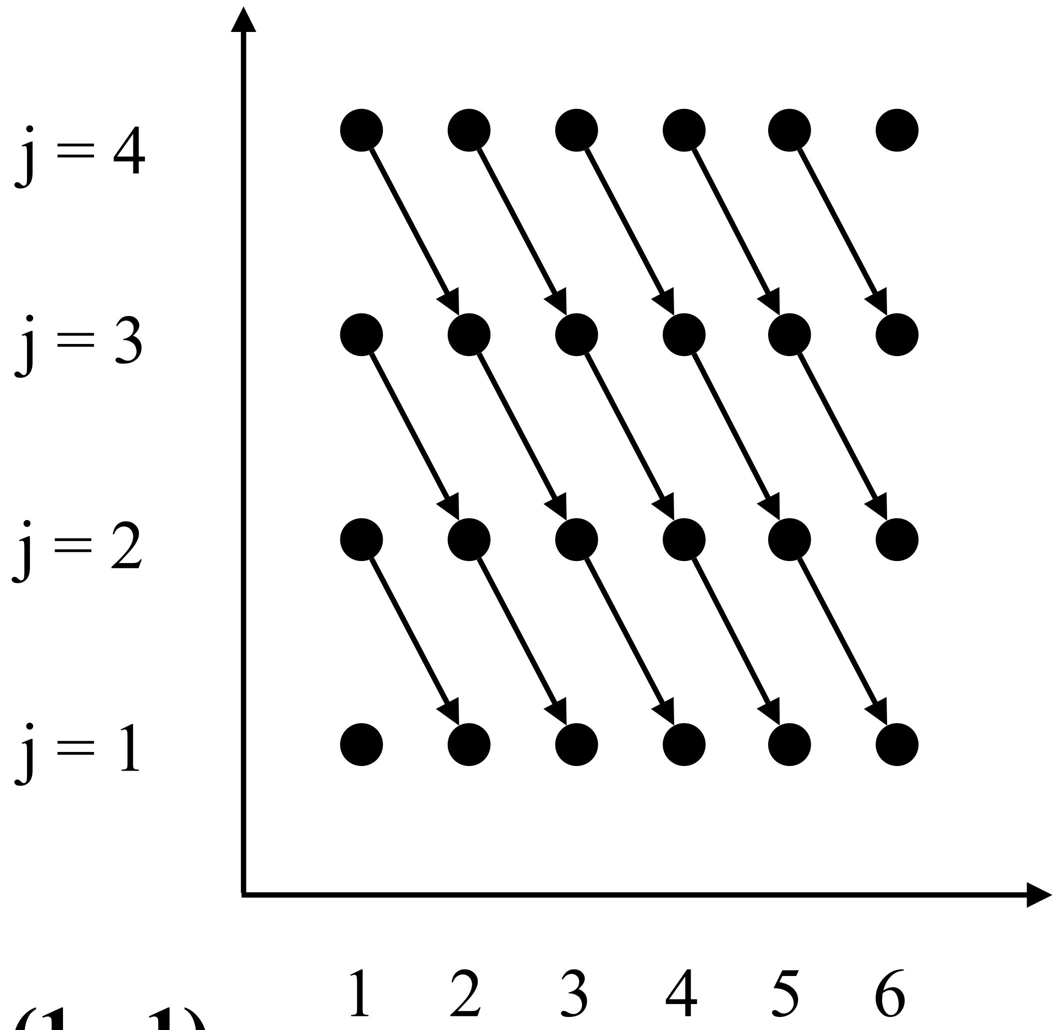
```
do I = 1, N
    do J = 1, N
    S1: A(I, J) = A(I+1, J-1)
```

Write After Read

Read in $S_1(1,2)$ to **Write** in $S_1(2,1)$

$S_1(i, j)$ to $S_1(i+1, j-1)$

Distance vector from read to write: (1, -1)



Processing Space: Affine Partition Schedule

- <**C**, **c**> to represent a partition
  - **C** is a *n by m* matrix
    - m = d (the loop level)
    - n is the dimension of the processor grid
  - **c** is a n-element constant vector
  - $\boldsymbol{p} = \mathbf{C} * \boldsymbol{i} + \mathbf{c}$

Notation:
bold fonts *for container variables;*
normal fonts for scalar variables.

- Examples

1-d processor grid

```
for (i=1; i<=N; i++)
    Y[i] = Z[i];
```

$\mathbf{C} = [1]$, $\mathbf{c} = [0]$, $p = i$

2-d processor grid

```
for (i=1; i<=N; i++)
   for (j=1; j<=N; j++)
      Y[i,j] = Z[i,j];
```

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$p = i$, $q = j$

Synchronization-free Parallelism

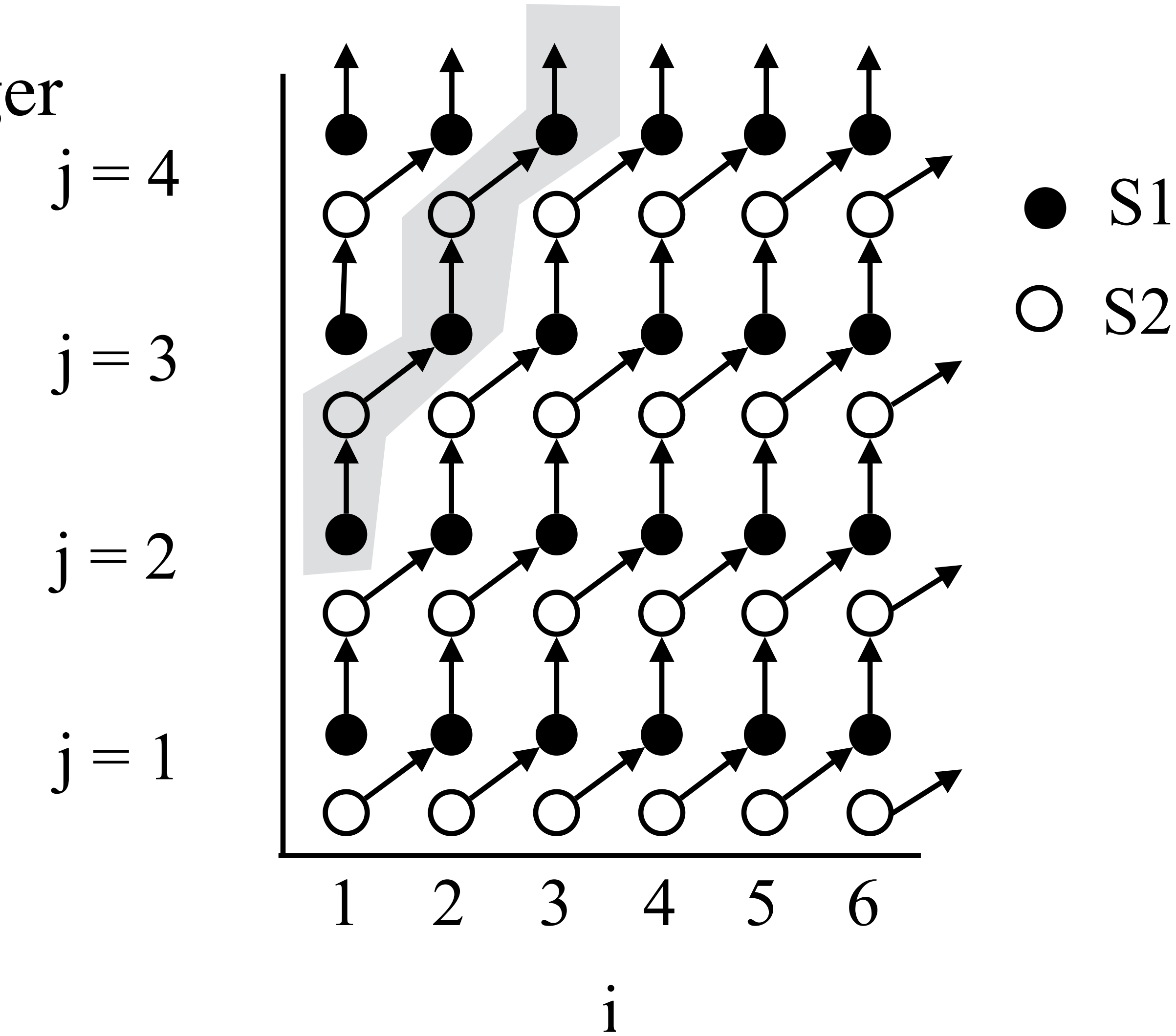
- Two memory references as <F_1, f_1, B_1, b_1> and <F_2, f_2, B_2, b_2>
- Let <$\mathbf{C_1}, \boldsymbol{c_1}$> and <$\mathbf{C_2}, \boldsymbol{c_2}$> represent their respective processor schedule
- To be synchronization-free
  - For all $\boldsymbol{i_1}$ in $\mathbf{Z_{d1}}$ (d1-dimension integer vectors) and $\boldsymbol{i_2}$ in $\mathbf{Z_{d2}}$ such that
    1. $\mathbf{B_1 * i_1 + b_1 >= 0}$, and
    2. $\mathbf{B_2 * i_2 + b_2 >= 0}$, and
    3. $\mathbf{F_1 * i_1 + f_1 = F_2 * i_2 + f_2}$, and
    4. It must be the case that $\mathbf{C_1 * i_1 + c_1 = C_2 * i_2 + c_2}$.

$\mathbf{F_1}$, f_1 is for memory reference, i.e., $\mathbf{F_1}*x + f_1$

$\mathbf{B_1}$, b_1 is for loop bound constraints, i.e., $\mathbf{B_1}*x + b_1$

Synchronization-free Parallelism

- To be synchronization-free
  - For all $\boldsymbol{i_1}$ in $\mathbf{Z_{d1}}$ (d1-dimension integer vectors) and $\boldsymbol{i_2}$ in $\mathbf{Z_{d2}}$ such that
    - $\mathbf{B_1} * \boldsymbol{i_1} + \boldsymbol{b_1} \mathbf{>= 0}$, and
    - $\mathbf{B_2} * \boldsymbol{i_2} + \boldsymbol{b_2} \mathbf{>= 0}$, and
    - $\mathbf{F_1} * \boldsymbol{i_1} + \boldsymbol{f_1} = \mathbf{F_2} * \boldsymbol{i_2} + \boldsymbol{f_2}$, and
    - It must be the case that $\mathbf{C_1} * \boldsymbol{i_1} + \boldsymbol{c_1} = \mathbf{C_2} * \boldsymbol{i_2} + \boldsymbol{c_2}$.




```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```

Synchronization-free Parallelism

```
for (i=1; i<=100; i++)
   for (j=1; j<=100; j++){
      S1: X[i,j] = X[i,j] + Y[i-1, j];
      S2: Y[i,j] = Y[i,j] + X[i, j-1];
   }
```

$1 <= i_1 <= 100$, $1 <= j_1 <= 100$,
$1 <= i_2 <= 100$, $1 <= j_2 <= 100$,
$i_1 = i_2$, $j_1 = j_2 - 1$,

$$[C_{11} \quad C_{12}]\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + [c_1] = [C_{21} \quad C_{22}]\begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + [c_2]$$

$$[C_{11} - C_{21} \quad C_{12} - C_{22}]\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + [c_1 - c_2 - C_{22}] = 0$$

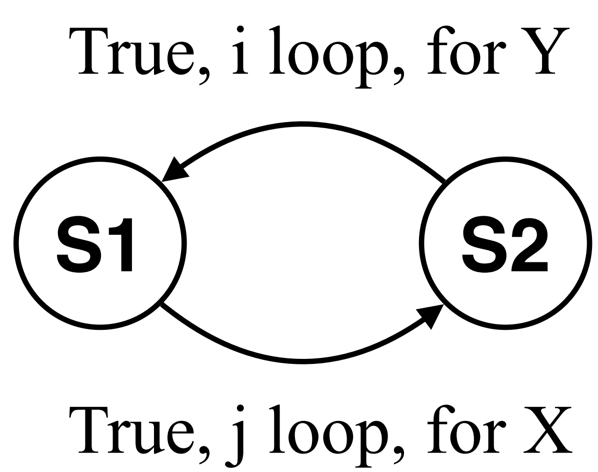
S1 to S2 dependence

$1 <= i_3 <= 100$, $1 <= j_3 <= 100$,
$1 <= i_4 <= 100$, $1 <= j_4 <= 100$,
$i_3 - 1 = i_4$, $j_3 = j_4$,

$$[C_{11} \quad C_{12}]\begin{bmatrix} i_3 \\ j_3 \end{bmatrix} + [c_1] = [C_{21} \quad C_{22}]\begin{bmatrix} i_4 \\ j_4 \end{bmatrix} + [c_2]$$

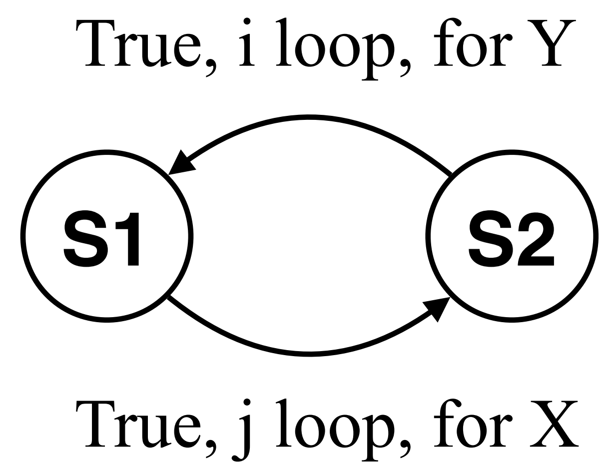
$$[C_{11} - C_{21} \quad C_{12} - C_{22}]\begin{bmatrix} i_3 \\ j_3 \end{bmatrix} + [c_1 - c_2 + C_{21}] = 0$$

S2 to S1 dependence



Synchronization-free Parallelism

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        S1: X[i,j] = X[i,j] + Y[i-1, j];
        S2: Y[i,j] = Y[i,j] + X[i, j-1];
    }
```



$$[C_{11} - C_{21} \quad C_{12} - C_{22}]\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + [c_1 - c_2 - C_{22}] = 0$$

$\boxed{C_{11}-C_{21} = 0,\ C_{12}-C_{22}=0,\ \&\ c_1-c_2-C_{22}=0}$

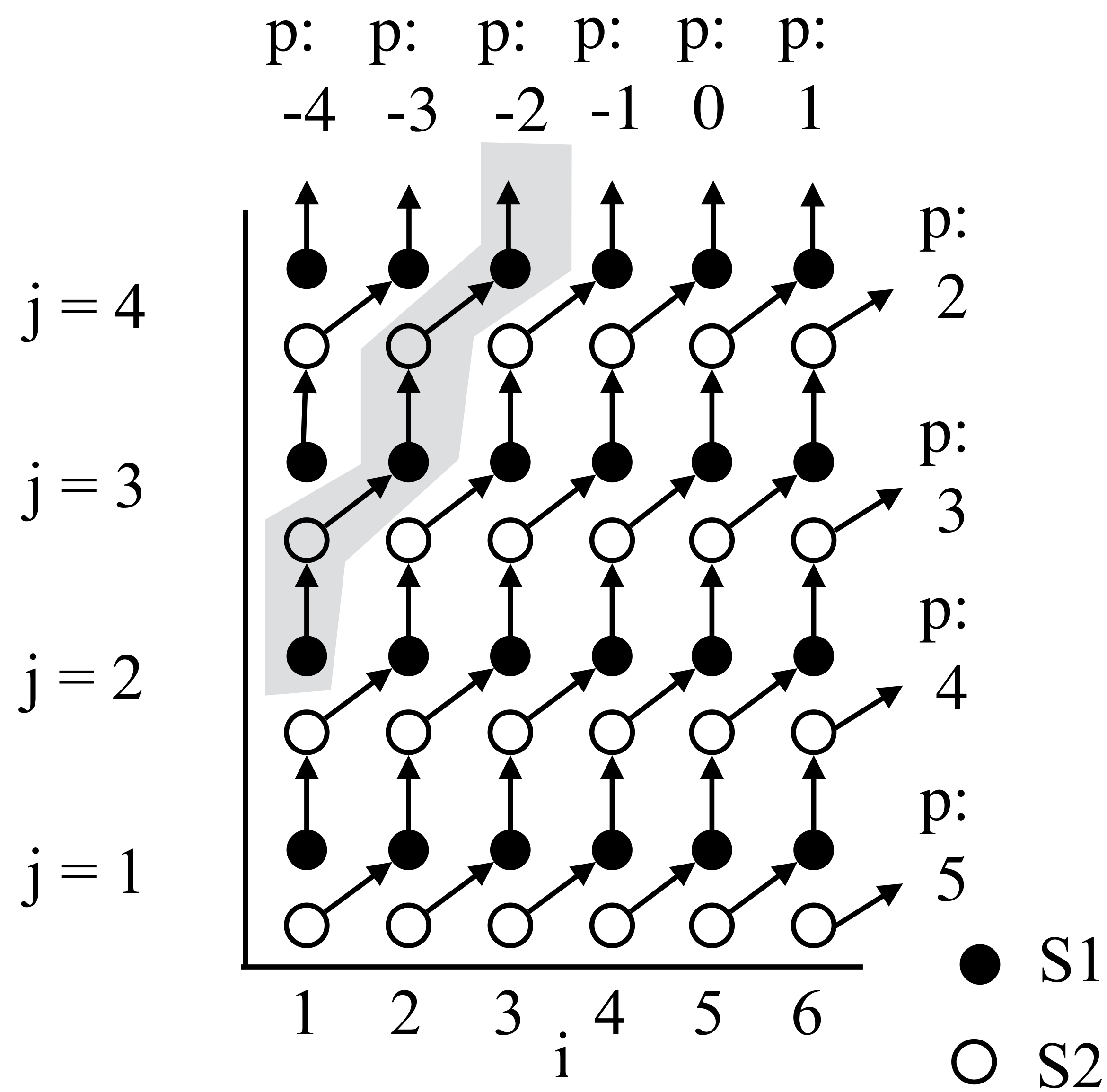
$$[C_{11} - C_{21} \quad C_{12} - C_{22}]\begin{bmatrix} i_3 \\ j_3 \end{bmatrix} + [c_1 - c_2 + C_{21}] = 0$$

$\boxed{C_{11}-C_{21} = 0,\ C_{12}-C_{22}=0,\ \&\ c_1-c_2+C_{21}=0}$

$C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1$

Solution

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
        Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
    }
```



p(S1): < [C_{11} C_{12}], [c_1]>

p(S2): < [C_{21} C_{22}], [c_2]>

Affine schedule for S1, p(S1): [C_{11} C_{12}] = [1 -1], c_1 = -1

i.e. (i,j) iteration of S1 to processor p = i-j-1;

Affine schedule for S2, p(S2) [C_{21} C_{22}]= [1 -1], c_2 =0

i.e. (i,j) iteration of S2 to processor p = i-j.

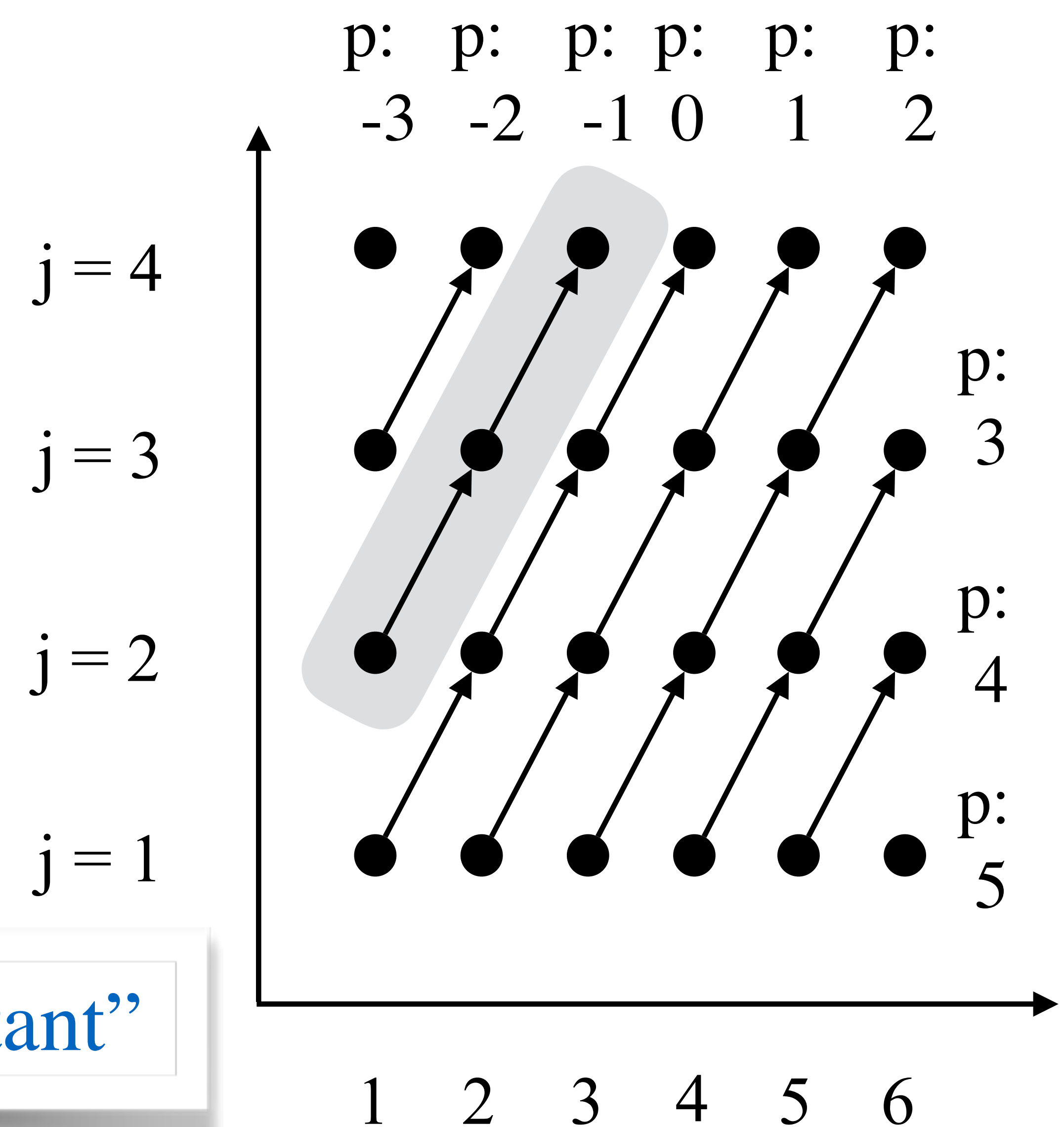
$$C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1$$

More Examples

Affine partition schedule

```
do I = 1, N
    do J = 1, N
      S1: A[I, J] = A[I-1, J - 1]
```

Read After Write



The hyperplane is j - i = "a constant"

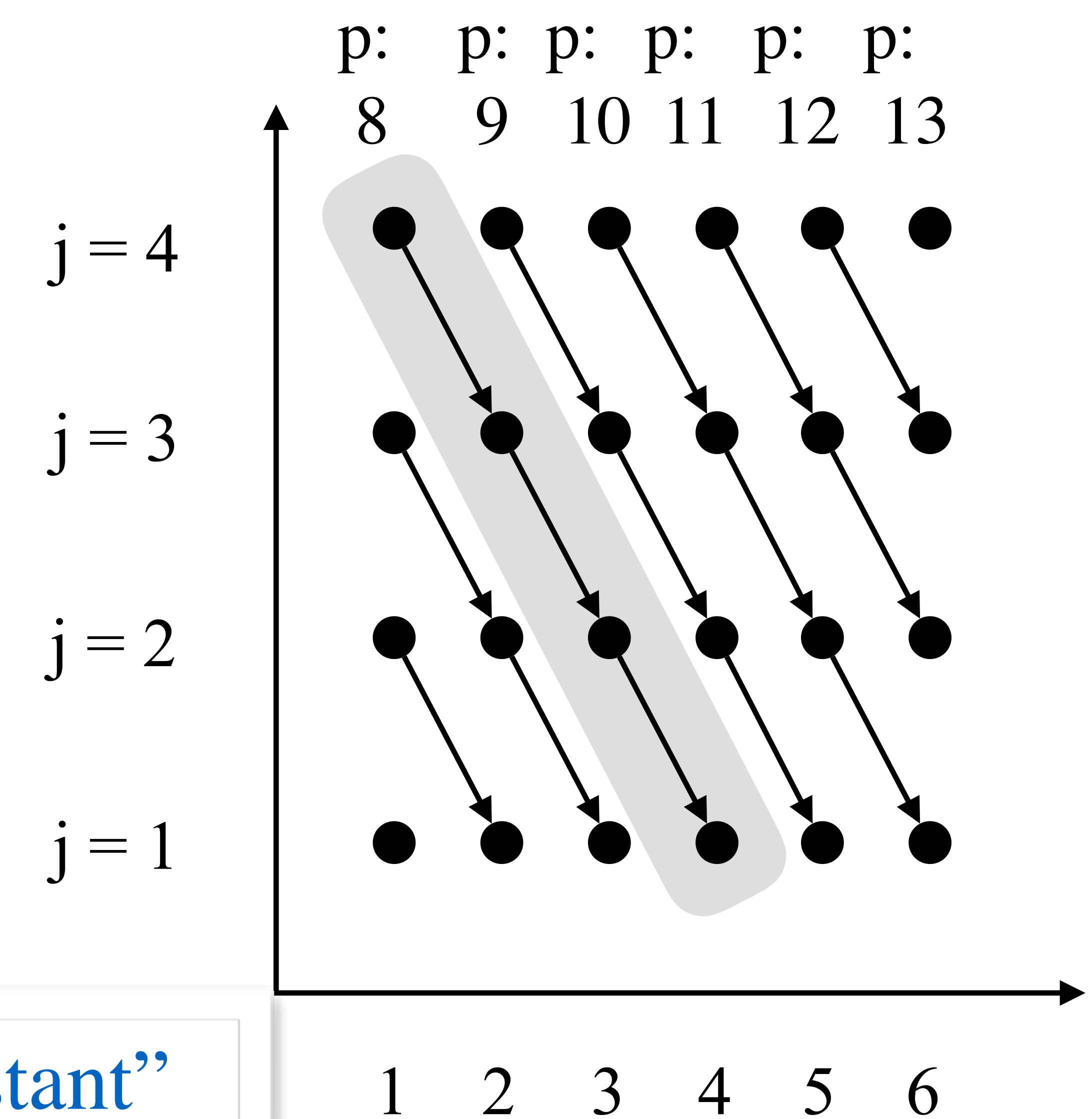
Affine schedule for S_1, $p(S_1)$: $C = [C_{11}\ C_{12}] = [1\ -1]$, $c = 0$

i.e. (i, j) iteration of S_1 to processor $p = i-j$;

More Examples

Affine partition schedule

```
do I = 1, N
    do J = 1, N
        S1: A[I, J] = A[I+1, J-1]
```



Write After Read

Read in $S_1(1,2)$ to **Write** in $S_1(2,1)$

$S_1(i, j)$ to $S_1(i+1, j-1)$

The hyperplane is j + i = "a constant"

Affine schedule for S1, p(S1): $C = [C_{11}\ C_{12}] = [1\ 1]$, $c = 0$

i.e. (i, j) iteration of S1 to processor p = i + j ;

Next Class

Reading

- ALSU, Chapter 11.1 - 11.7