CS 314 Principles of Programming Languages

Lecture 20: Parallelism and Dependence Analysis

Prof. Zheng Zhang

Rutgers University

November 16, 2018

Class Information

- Homework 7 is released.
- Project 2 deadline will be extended to 11/25 Sunday.

Review: Dependence Test

Given

do $i_1 = L_1, U_1$

 ...

  do $i_n = L_n, U_n$

   S1 : $\mathbf{A[\ f_1(\ i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)\ ]} = \ldots$

   S2 : $\ldots = \mathbf{A[\ g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)\ ]}$

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.

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

Integer Linear Programming (ILP) for Dependence Test

Does ∃ 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];
    }
```


Consider the two memory references:

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

α: (i_1, j_1)

β: (i_2, j_2)

Access the same memory location →

$i_1 = i_2$

$j_1 = j_2 - 1$

Loop bounds constraint →

$1 \le i_1 \le 100$

$1 \le j_1 \le 100$

$1 \le i_2 \le 100$

$1 \le j_2 \le 100$

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

Integer Linear Programming (ILP) for Dependence Test

If we use the matrix vector notation <F_1, f_1, B_1, b_1> and <F_2, f_2, B_2, b_2> for two references at two iterations α: (i_1, j_1) and β: (i_2, j_2)

$$\begin{aligned} i_1 &= i_2 \\ j_1 &= j_2 - 1 \\ 1 &<= i_1 <= 100 \\ 1 &<= j_1 <= 100 \\ 1 &<= i_2 <= 100 \\ 1 &<= j_2 <= 100 \end{aligned}$$

Memory reference **X[i_1, j_1]**

$$\mathbf{F_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad f_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} i_1 \\ j_1 \end{bmatrix}$$

Memory reference **X[i_2, j_2-1]**

$$\mathbf{F_1} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + f_1 = \mathbf{F_2} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + f_2$$

$$\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} = \begin{bmatrix} i_2 \\ j_2 - 1 \end{bmatrix}$$

$$\mathbf{F_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad f_2 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} i_1 \\ j_2 - 1 \end{bmatrix}$$

Integer Linear Programming (ILP) for Dependence Test

If we use the matrix vector notation <F_1, f_1, B_1, b_1> and <F_2, f_2, B_2, b_2> for two references at two iterations α: (i_1, j_1) and β: (i_2, j_2)

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

Loop bounds for ($\mathbf{i_1, j_1}$)

$$B_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \quad b_1 = \begin{bmatrix} -1 \\ -1 \\ 100 \\ 100 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \\ 100 \\ 100 \end{bmatrix} >= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$B_1 \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + b_1 >= 0$$

$$B_2 \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + b_2 >= 0$$

Loop bounds for ($\mathbf{i_2, j_2}$)

$$B_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \quad b_2 = \begin{bmatrix} -1 \\ -1 \\ 100 \\ 100 \end{bmatrix}$$

Putting Everything Together

If we use the matrix vector notation <F_1, f_1, B_1, b_1> and <F_2, f_2, B_2, b_2> for two references at two iterations α: (i_1, j_1) and β: (i_2, j_2)

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

$$\mathbf{F_1}\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + f_1 = \mathbf{F_2}\begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + f_2$$

$$B_1\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + b_1 >= 0$$

$$B_2\begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + b_2 >= 0$$

$$\mathbf{F_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad f_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \mathbf{F_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad f_2 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

B_1 , b_1 , B_2 , b_2 **see previous slides**

Review: Parallelizing Affine Loops

Three spaces

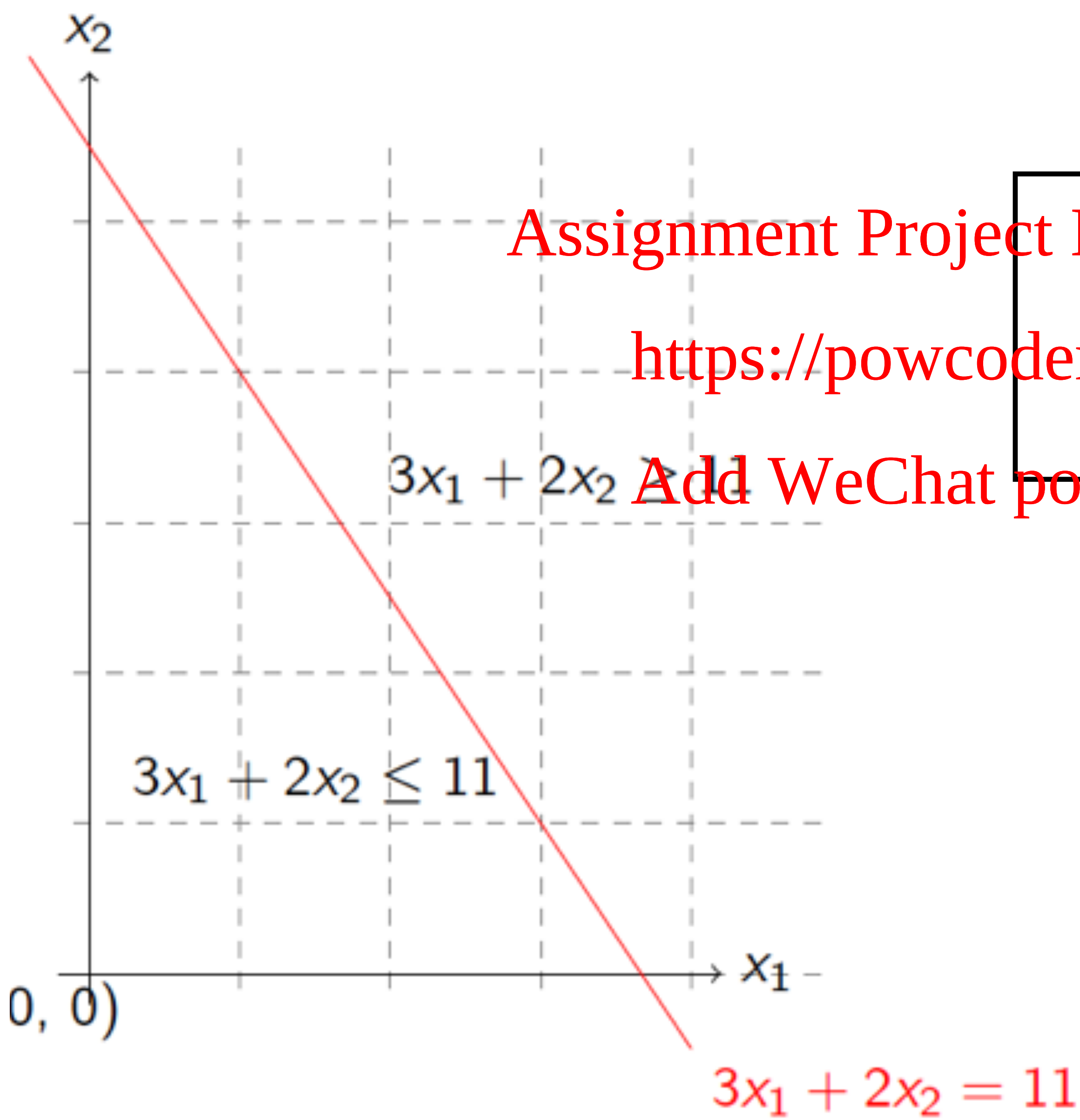
- Iteration space
  - The set of dynamic execution instances
    For instance, the set of value vectors taken by loop indices
  - A k-dimensional space for a k-level loop nest
- Data space
  - The set of array elements accessed
  - An n-dimensional space for an n-dimensional array
- Processor space
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

Affine Half Space

Definition

An affine half-space of Z^d is defined as the set of points

$$\{ \vec{x} \in Z^d \mid \vec{a} * \vec{x} <= b \}$$



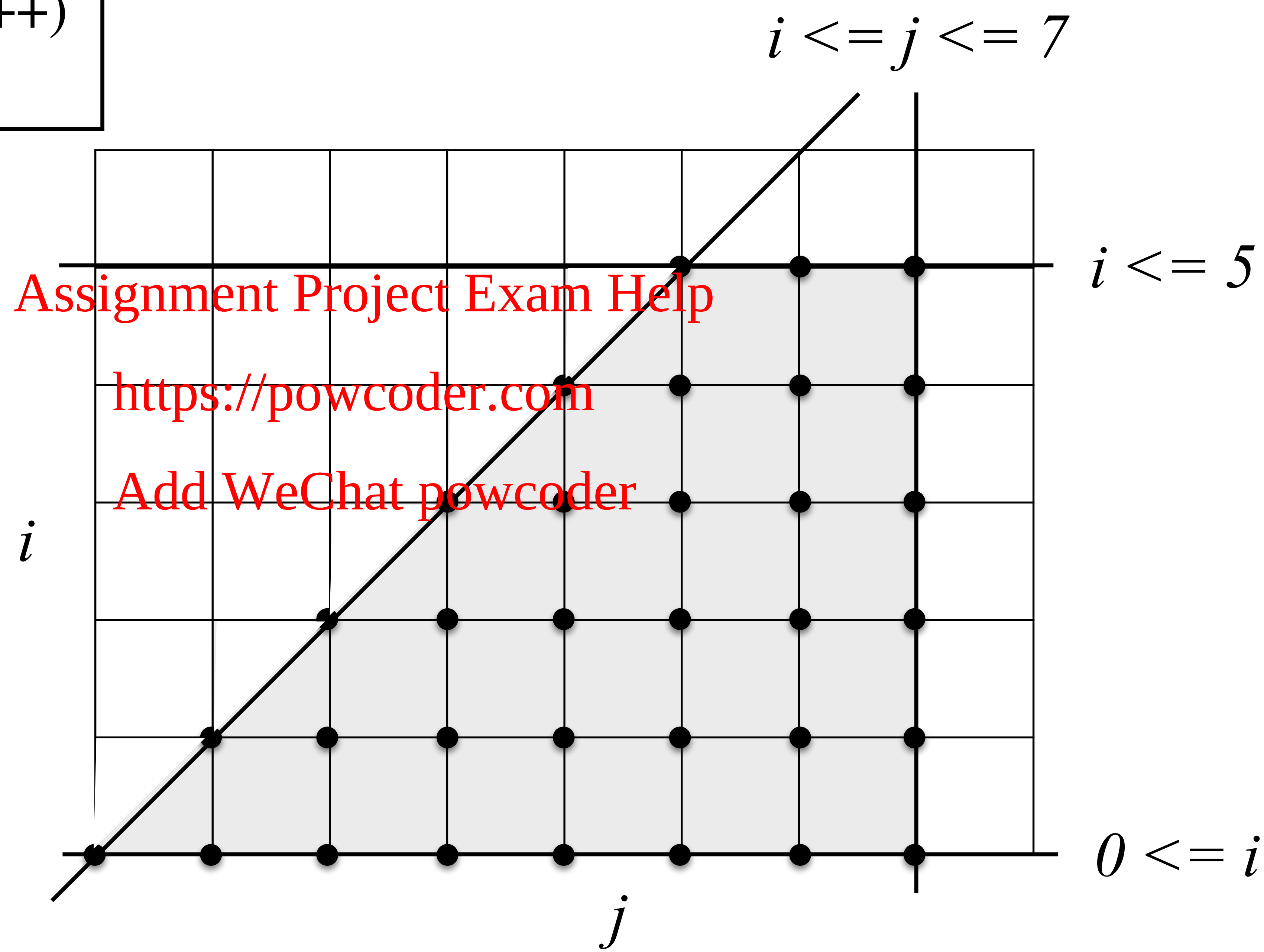
$\vec{a} * \vec{x} = b$ is the hyperplane that divides the d-dimensional space into two half-spaces.

Iteration Space

- Bounded by multiple hyperplanes

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

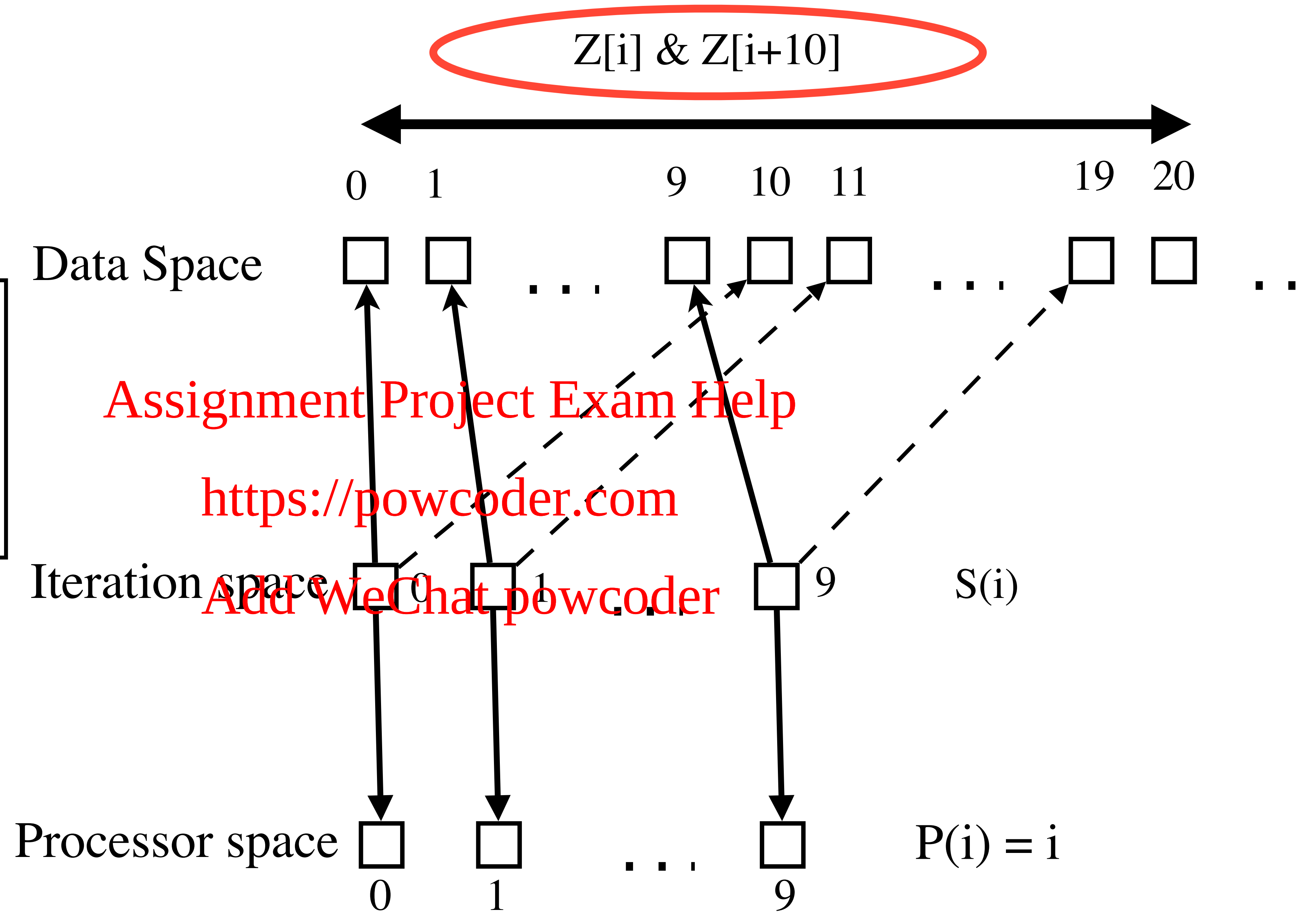
$0 <= i <= 5$
$i <= j <= 7$



Three Spaces

- **Iteration space, data space, and processor space**

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



Synchronization-free Parallelism

Parallelize an application **without** allowing any *communication* or *synchronization* among (logical) processors.

Example:

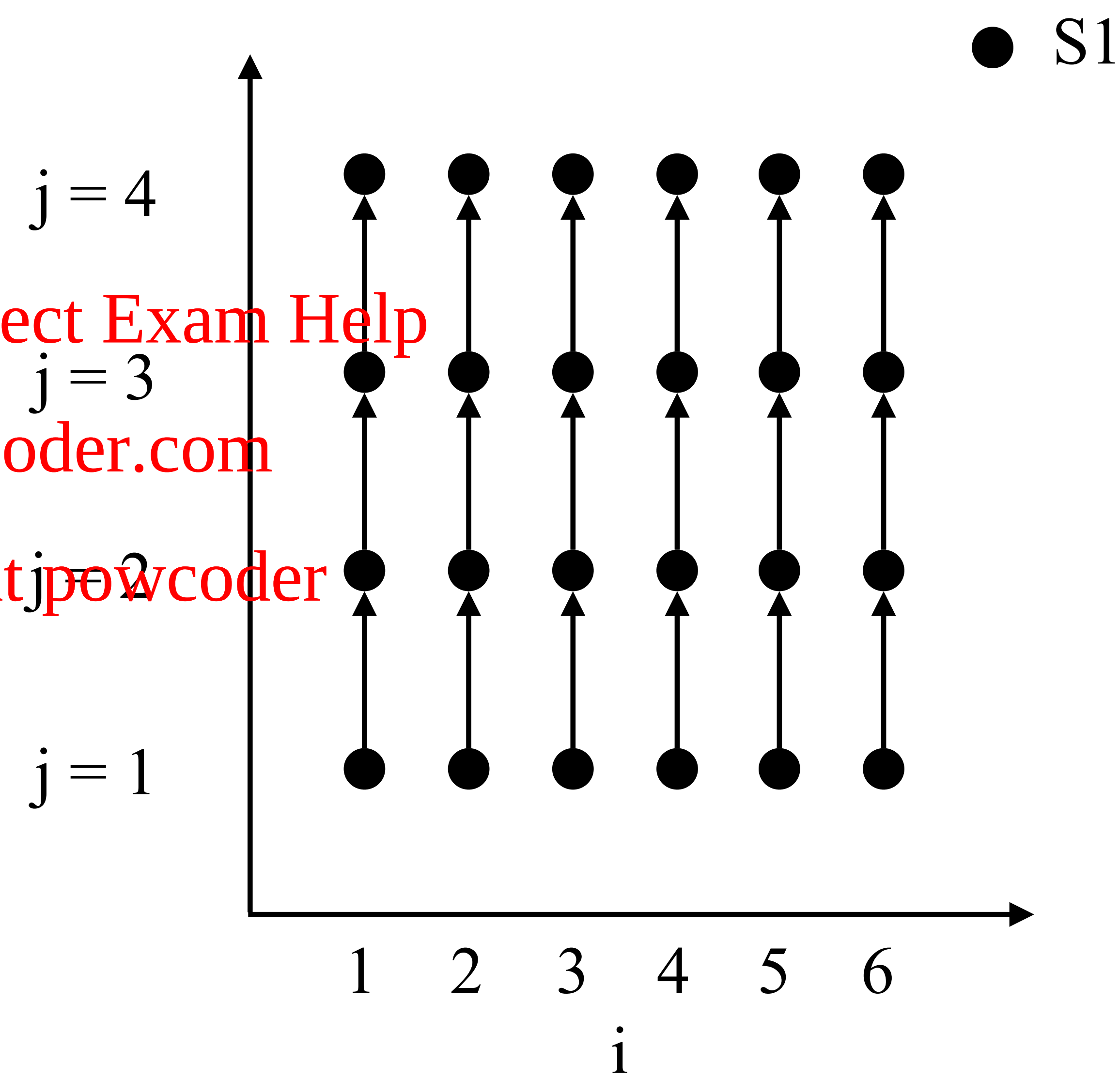
```
do i = 1, N
    do j = 1, N
        S1: A[i, j] = A[i, j - 1]
```


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

↓

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

Dependence from **S1** to **S1**

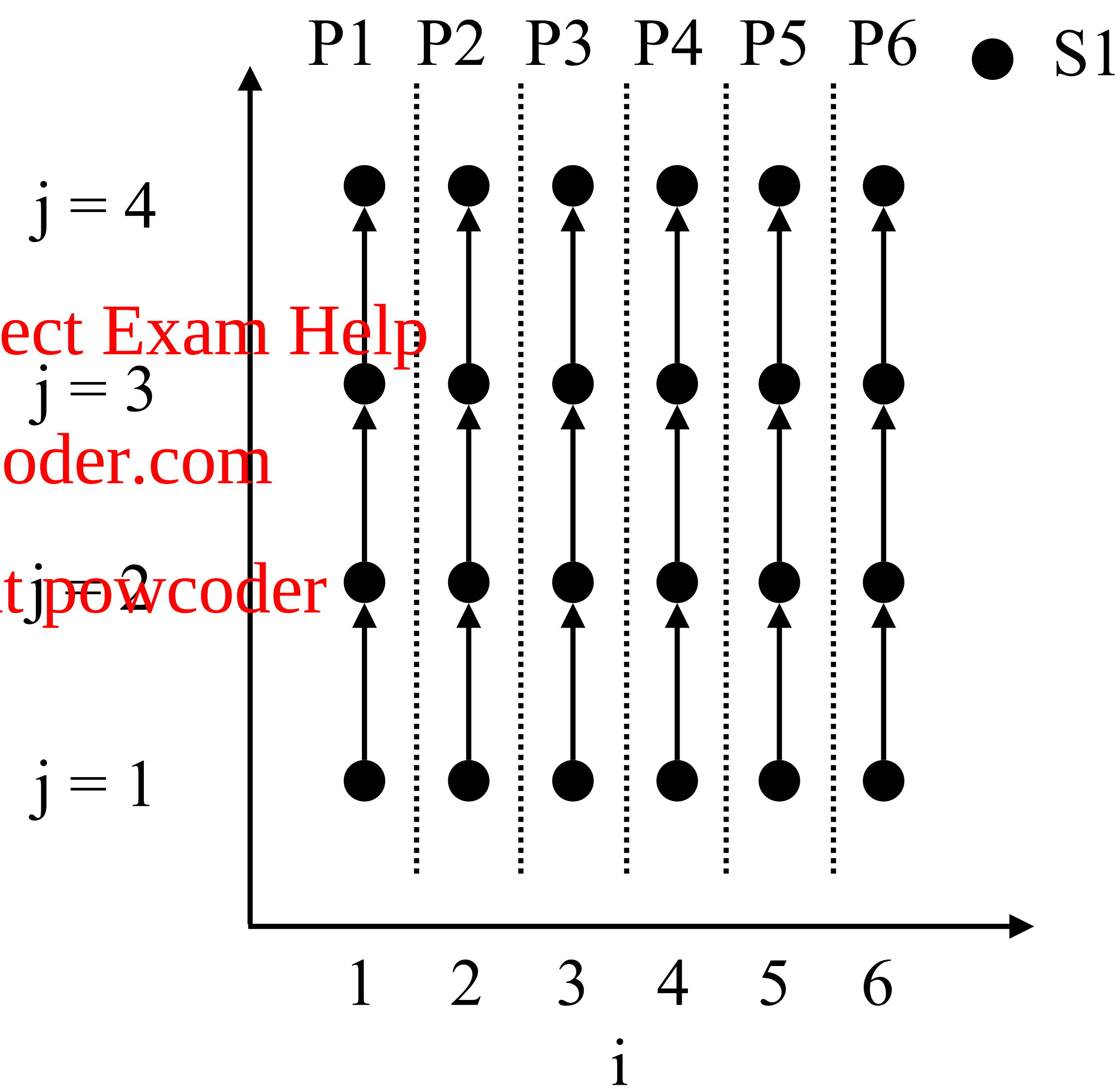


Synchronization-free Parallelism

Parallelize an application **without** allowing any *communication* or *synchronization* among (logical) processors.

Example:

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

Dependence from **S1** to **S1**

Communication is limited to the iterations on one processor.

Synchronization-free Parallelism

Parallelize an application **without** allowing any *communication* or *synchronization* among (logical) processors.

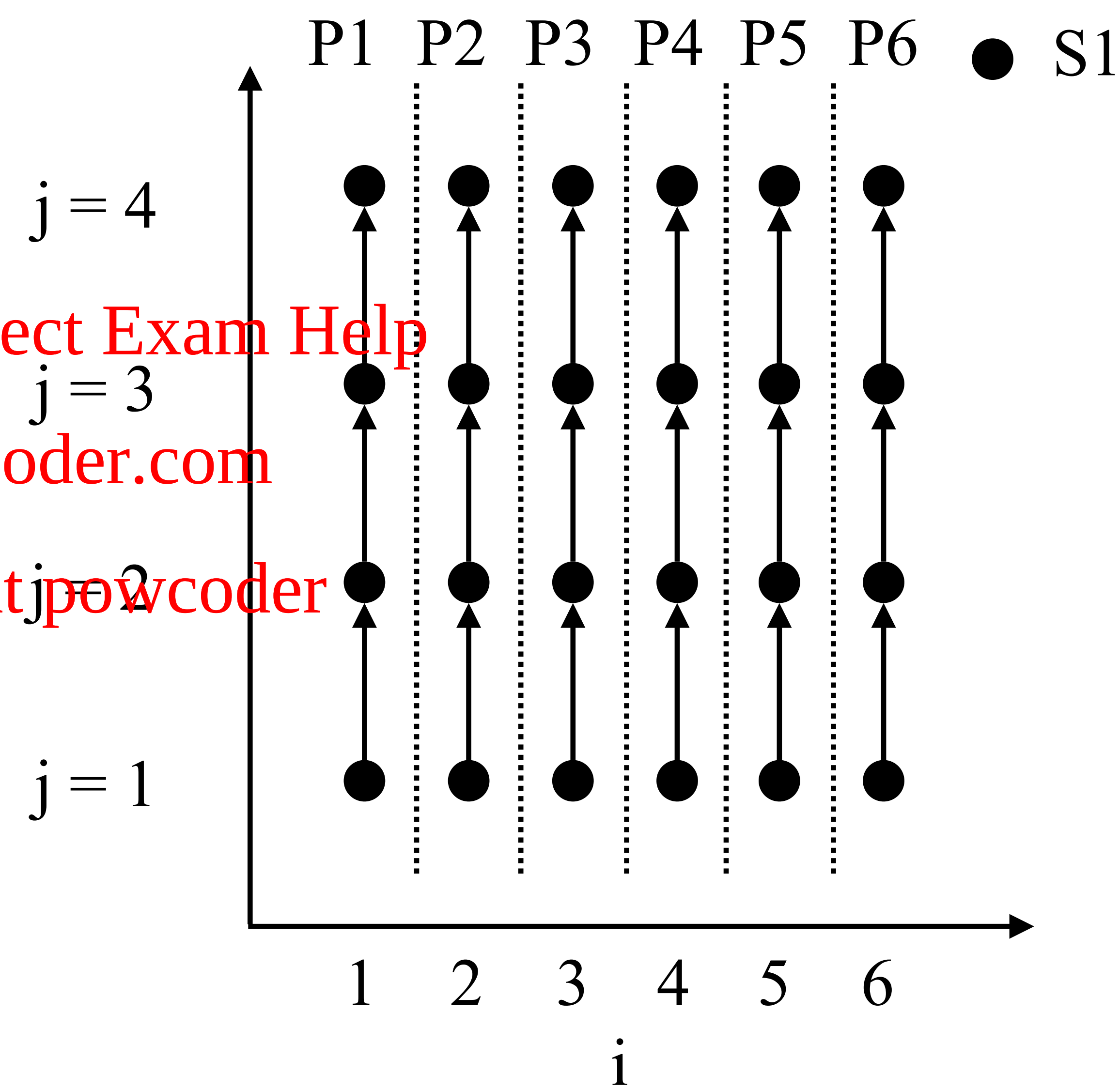
Example:

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



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

Answer: the "i" loop

Synchronization-free Parallelism

Parallelize an application **without** allowing any *communication* or *synchronization* among (logical) processors.

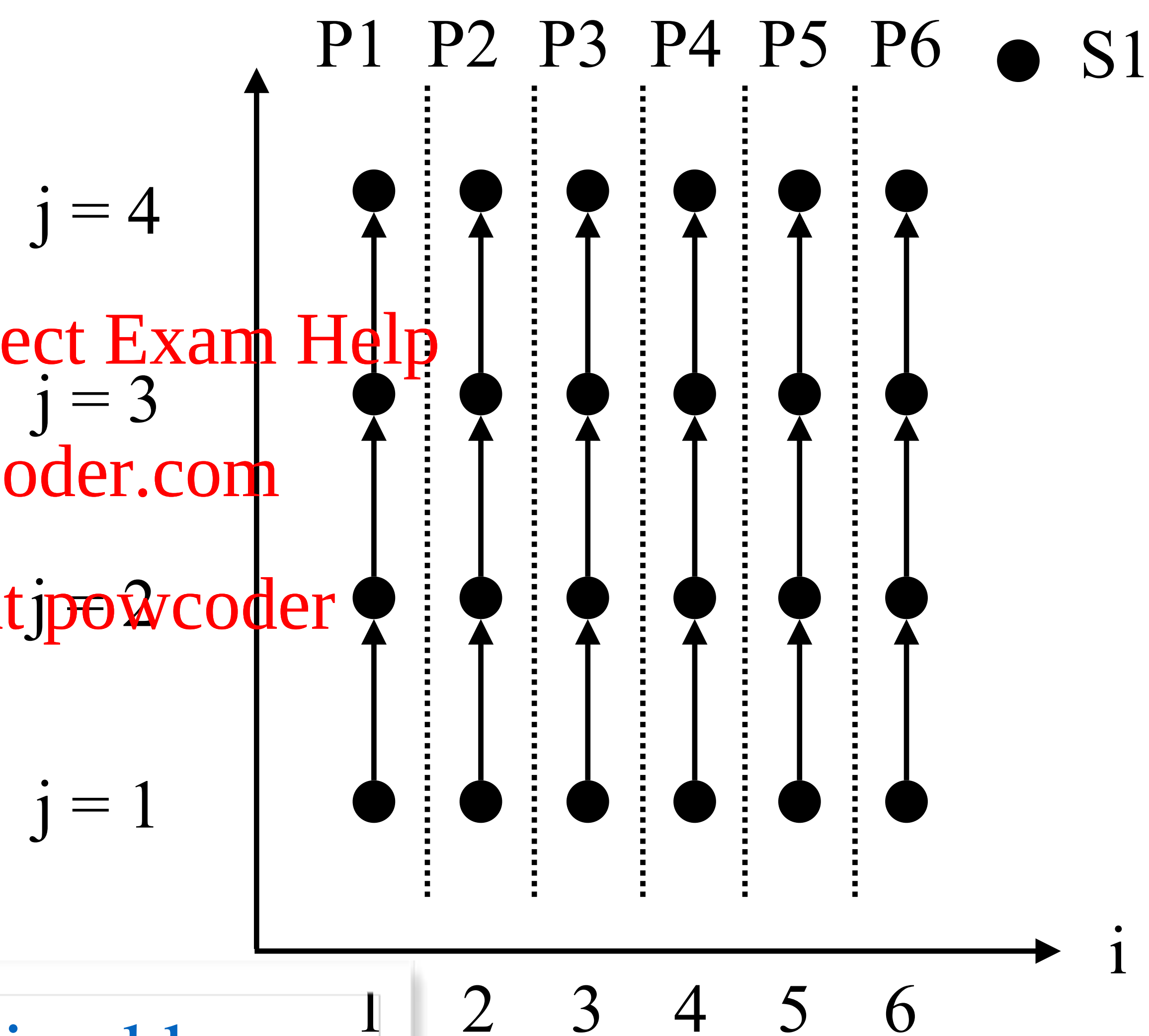
Example 1:

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



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

The dependence chain is characterized by a **hyperplane**. In this case it is "i = constant".

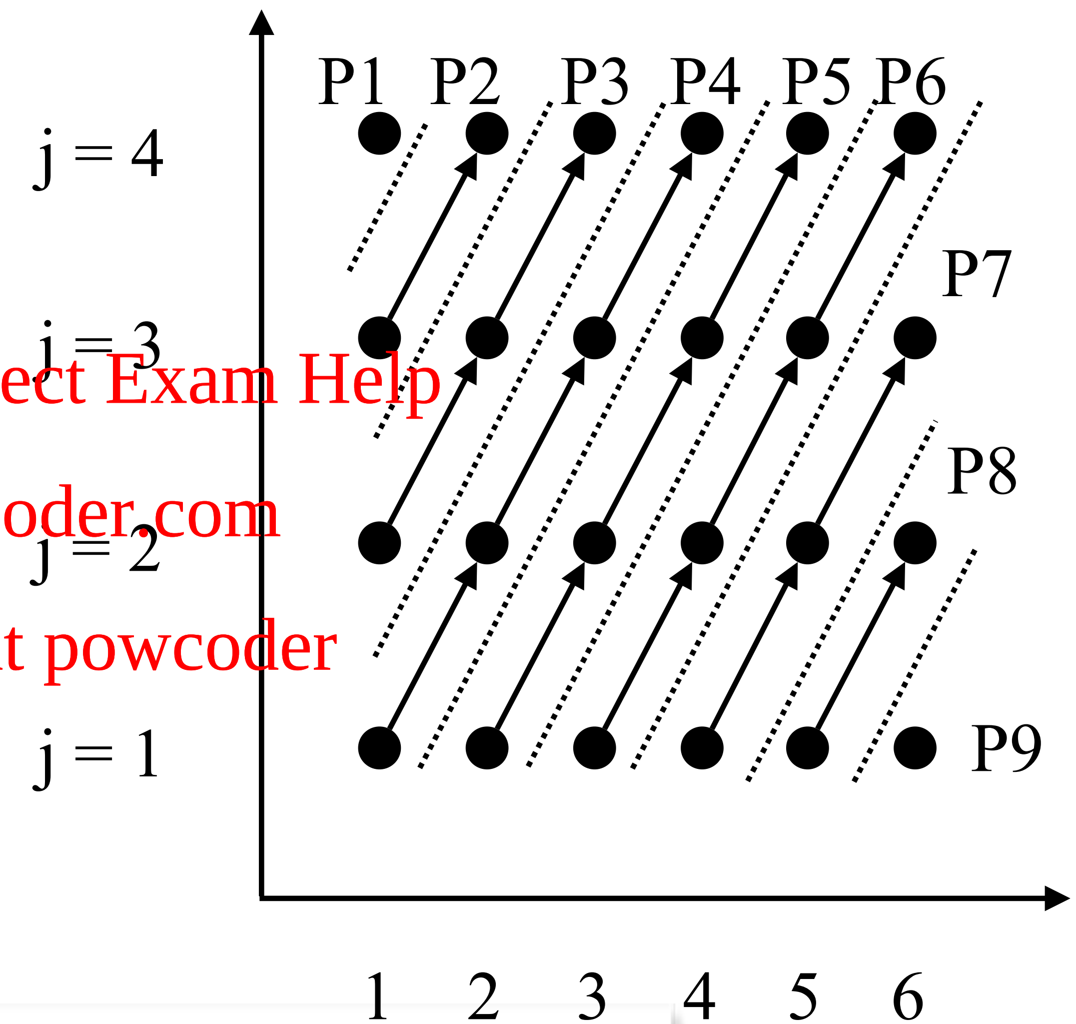
Synchronization-free Parallelism

Parallelize an application **without** allowing any *communication* or *synchronization* among (logical) processors.

Example 2:

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


Dependence from **S1**(i,j) to **S1**(i+1,j+1)



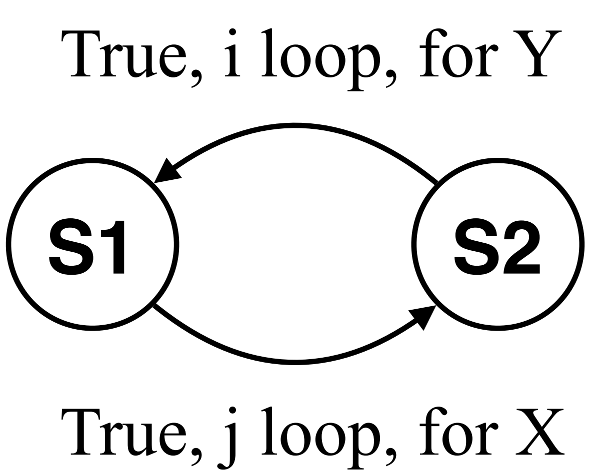
The dependence chain is characterized by a **hyperplane**. In this case it is "j - i + constant = 0".

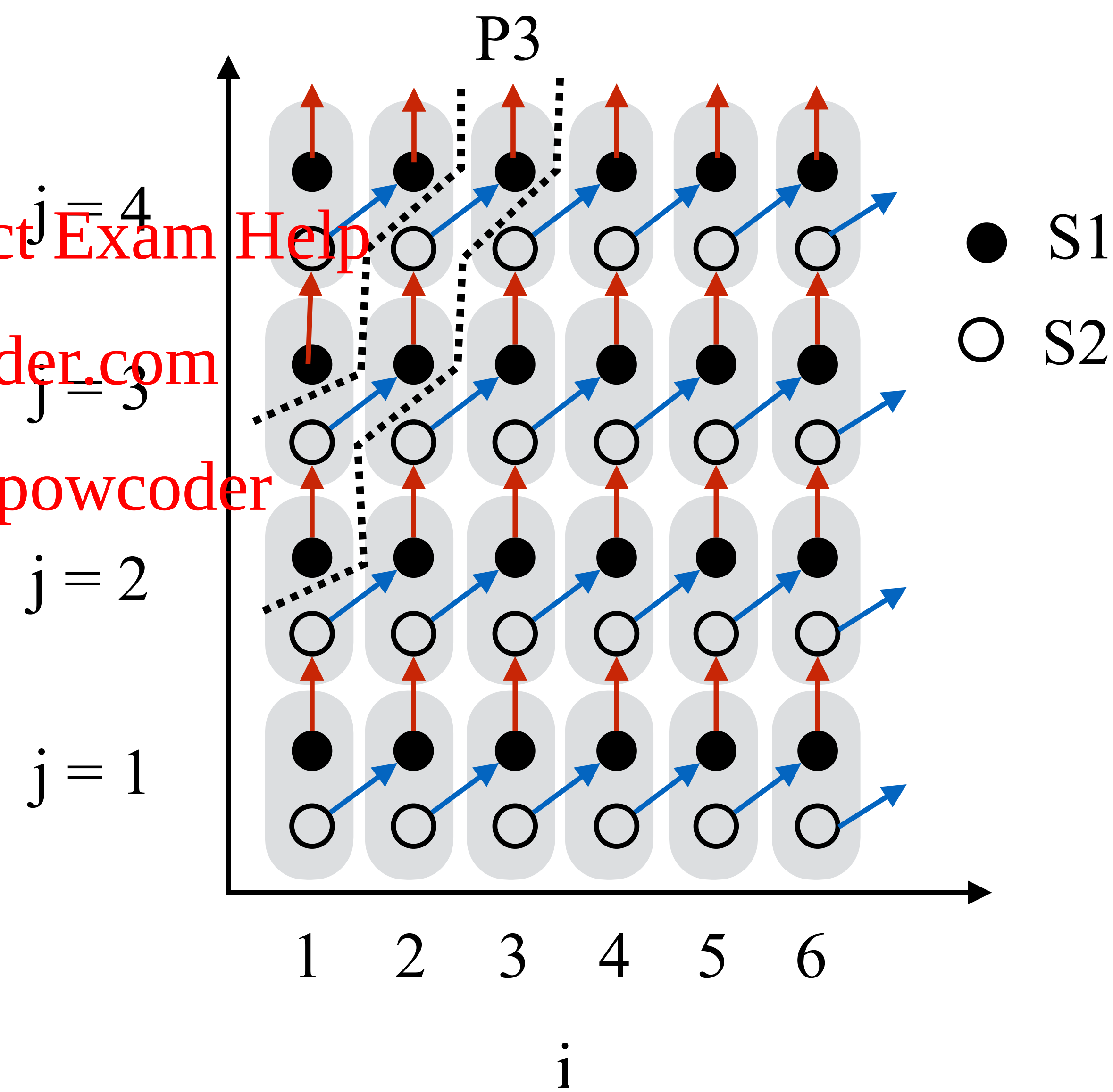
Synchronization-free Parallelism

Parallelize an application **without** allowing any *communication* or *synchronization* among (logical) processors.

Example 3:

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

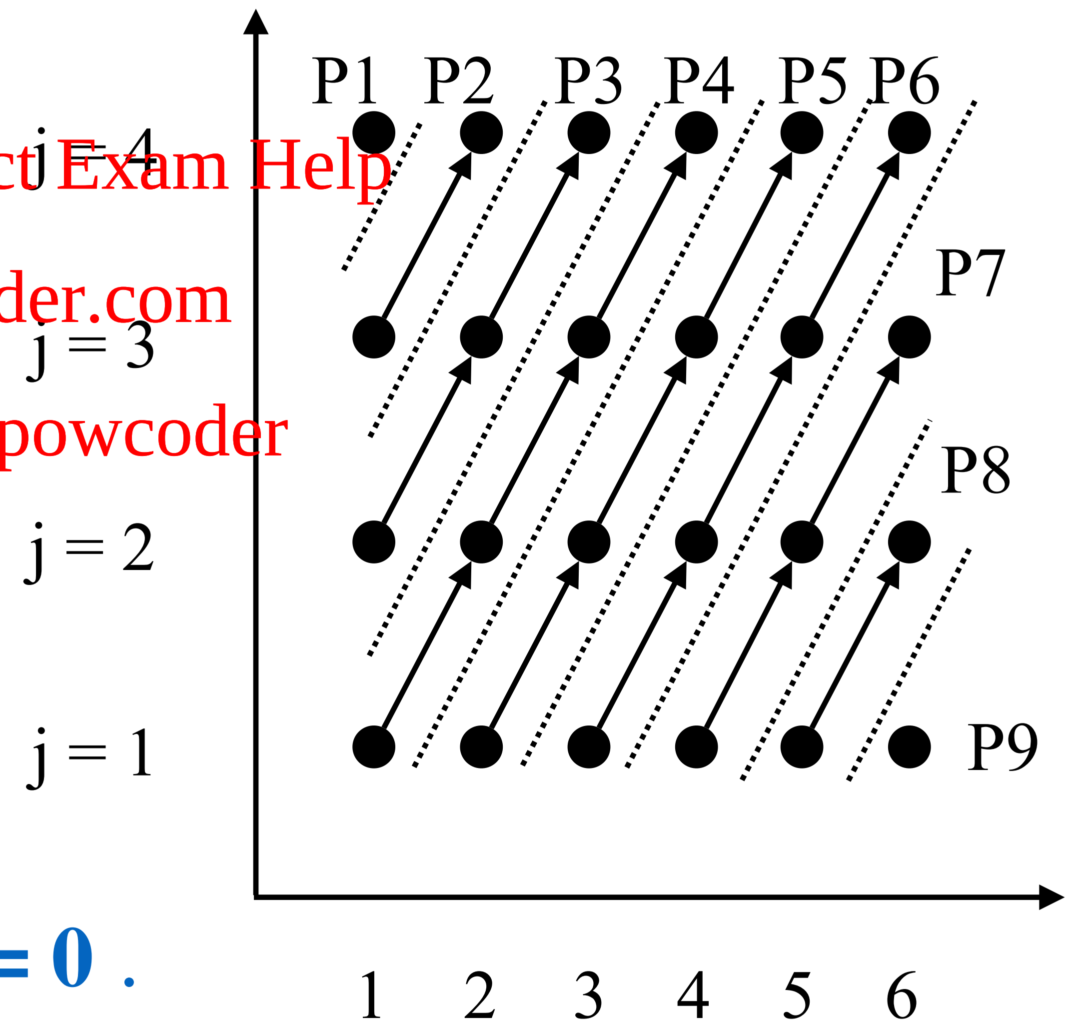
Dependence from **S2**(1,1) to **S1**(2,1)

Dependence and Parallelization

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

<u>Example</u>:


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



Dependence from **S1**(i,j) to **S1**(i+1,j+1)

The hyperplane is **j - i + constant = 0** .

Processing Space: Affine Partition Schedule

- Map an iteration to a processor using $<\mathbf{C}, d>$

  C is a *n by m* matrix

  - m = d (the loop level)
  - n is the dimension of the processor grid

  $\vec{d}$ is a n-element constant vector

  $\vec{p} = \mathbf{C}\,\vec{x} + \vec{d}$**, where** $\vec{x}$ **is an iteration vector**

Processing Space: Affine Partition Schedule

- Map an iteration to a processor using < **C**, d >

$$\vec{p} = \mathbf{C}\,\vec{x} + \vec{d}, \text{ where } \vec{x} \text{ is an iteration vector}$$

- Example

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

$\mathbf{C} = [1]$, $\mathbf{d} = [0]$

$\vec{p}(\ S(i)\ ) = 1{*}i + 0$

$= i$

Map iteration **i** to Processor **i**

Synchronization-free Parallelism

- Two memory references as $<F_1, f_1, B_1, b_1>$ and $<F_2, f_2, B_2, b_2>$ such that $<F_2, f_2, B_2, b_2>$ at iteration α: (i_1, j_1) depends on $<F_1, f_1, B_1, b_1>$ at iteration β: (i_2, j_2)
  - $\mathbf{F_1}$ is a matrix and f_1 is a vector.
    The affine memory access index is $\mathbf{F_1} * \alpha + f_1$.
  - $\mathbf{B_1}$ is a matrix and b_1 is a vector.
    The affine loop bounds can be expressed as $\mathbf{B_1} * \alpha + b_1 >= 0$

- Let $<\mathbf{C_1}, \mathbf{d_1}>$ and $<\mathbf{C_2}, \mathbf{d_2}>$ represent the respective processor schedule, to have synchronization-free parallelism,

$$\mathbf{C_1 * \alpha + d_1 = C_2 * \beta + d_2}$$

These two memory references must execute on the same processor (sequentially).

Synchronization-free Parallelism

- Two memory references as $\langle F_1, f_1, B_1, b_1 \rangle$ and $\langle F_2, f_2, B_2, b_2 \rangle$ such that $\langle F_2, f_2, B_2, b_2 \rangle$ at iteration (i_1, j_1) depends on $\langle F_1, f_1, B_1, b_1 \rangle$ at iteration (i_2, j_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];
    }
```


$$\mathbf{F_1} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + f_1 = \mathbf{F_2} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + f_2$$

$$B_1 \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + b_1 >= 0$$

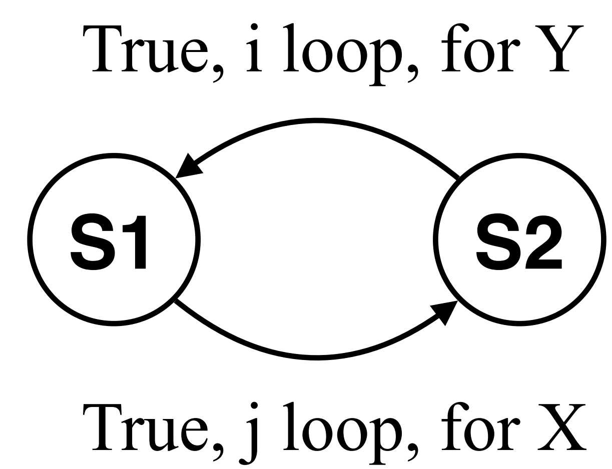
$$B_2 \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + b_2 >= 0$$

- We want to find processor schedule $\langle \mathbf{C_1}, \mathbf{d_1} \rangle$ and $\langle \mathbf{C_2}, \mathbf{d_2} \rangle$ such that

$$[\, C_{11}\ C_{12} \,] \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + [\, d_1 \,] = [\, C_{21}\ C_{22} \,] \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + [\, d_2 \,]$$

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];
    }
```



$1<=i_1<=100, \quad 1<=j_1<=100,$

$1<=i_2<=100, \quad 1<=j_2<=100,$

$i_1 = i_2, \quad j_1 = j_2 - 1,$

$$[C_{11}\ C_{12}]\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + [d_1] = [C_{11}\ C_{12}]\begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + [d_2]$$

$$\mathbf{C_{11} - C_{21} = 0}$$

$$\mathbf{C_{12} - C_{22} = 0}$$

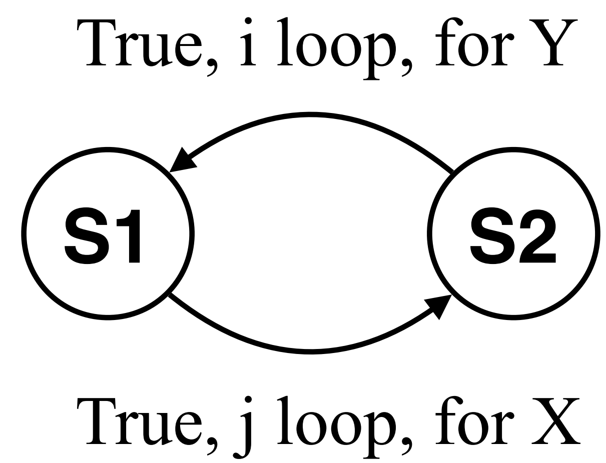
$$\mathbf{d_1 - d_2 - C_{22} = 0}$$

$$[C_{11} - C_{21}, C_{12} - C_{22}]\begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + [d_1 - d_2 - C_{22}] = 0$$

S1 to S2 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];
    }
```



$1 <= i_3 <= 100, \quad 1 <= j_3 <= 100,$

$1 <= i_4 <= 100, \quad 1 <= j_3 <= 100,$

$i_3 - 1 = i_4, \quad j_3 = j_4,$

$$[C_{11}\ C_{12}]\begin{bmatrix} i_3 \\ j_3 \end{bmatrix} + [d_1] = [C_{11}\ C_{12}]\begin{bmatrix} i_4 \\ j_4 \end{bmatrix} + [d_2]$$

$$\mathbf{C_{11} - C_{21} = 0}$$

$$\mathbf{C_{12} - C_{22} = 0}$$

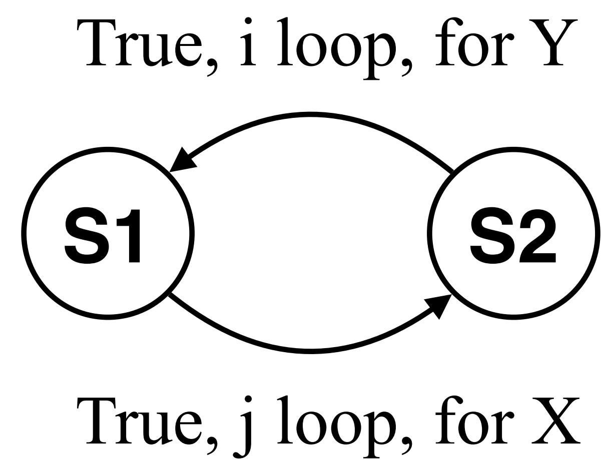
$$\mathbf{d_1 - d_2 + C_{21} = 0}$$

$$[C_{11} - C_{21}, C_{12} - C_{22}]\begin{bmatrix} i_3 \\ j_3 \end{bmatrix} + [d_1 - d_2 + C_{21}] = 0$$

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];
    }
```



S1 to S2 dependence

$$\mathbf{C_{11} - C_{21} = 0}$$
$$\mathbf{C_{12} - C_{22} = 0}$$
$$\mathbf{d_1 - d_2 - C_{22} = 0}$$

S2 to S1 dependence

$$\mathbf{C_{11} - C_{21} = 0}$$
$$\mathbf{C_{12} - C_{22} = 0}$$
$$\mathbf{d_1 - d_2 + C_{21} = 0}$$

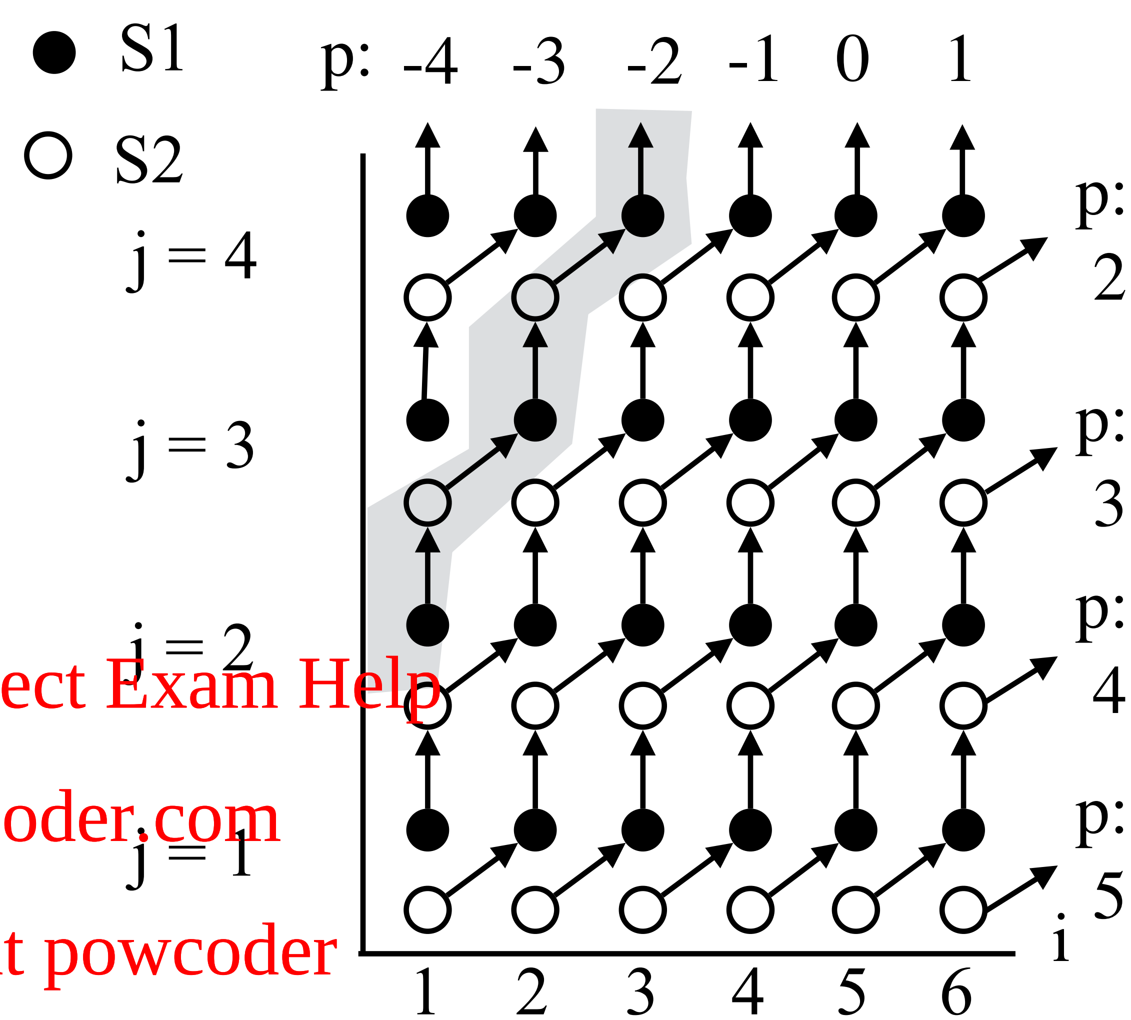
$$C_{11} = C_{21} = -C_{22} = -C_{12} = d_2 - d_1$$

Synchronization-free Parallelism

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];
   }
```

$$\mathbf{C_{11} = C_{21} = -C_{22} = -C_{12} = d_2 - d_1}$$



One Potential Solution:

Affine schedule for S1, p(S1): $[C_{11}\ C_{12}] = [1\ -1]$, $d_1 = -1$

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

Affine schedule for S2, p(S2): $[C_{21}\ C_{22}] = [1\ -1]$, $d_2 = 0$

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

Code Generation

```
for (i=1; i<=6; i++)
    for (j=1; j<=4; j++){
        X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
        Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
    }
```

S1(i, j): processor p = i-j-1;
S2(i, j): processor p = i-j.


```
forall (p=-4; p<=5; p++)
    for (i=1; i<=6; i++)
        for (j=1; j<=4; j++){
            if (p== i-j-1)
                X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
            if (p== i-j)
                Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
        }
```

- Step 1: find processor ID ranges
  - S1: $-4 <= p <= 4$
  - S2: $-3 <= p <= 5$
  - Union: $-4 <= p <= 5$
- Step 2: generate code

Naive Code Generation

```
forall (p=-4; p<=5; p++)
    for (i=1; i<=6; i++)
        for (j=1; j<=4; j++){
            if (p== i-j-1)
                X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
            if (p== i-j)
                Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
        }
```


What are the issues with this code?

- Idle iterations
- Lots of tests in loop body

Remove Idle Iterations

Some iterations have idle operations

For example, when p=-4, only 1 of the 24 iterations has useful operations, i = 1, j =4 .

```
forall (p=-4; p<=5; p++)
   for (i=1; i<=6; i++)
      for (j=1; j<=4; j++){
         if (p== i-j-1)
            X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
         if (p== i-j)
            Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
      }
```

Make Loop Bounds Tighter

-4 <= p <= 5 1<= i <= 6 1<= j <= 4 i-p-1=j	-4 <= p <= 5 1<= i <= 6 1<= j <= 4 i-p=j

↓ Fourier-Motzkin Elimination ↓

S1

j:	i-p-1<= j <= i-p-1
	1<=j <= 4
i:	p+2<=i <= p+5 ← Eliminate j
	1<=i <= 6
p:	-4<= p<= 4 ← Eliminate i

S2

j:	i-p<=j <= i-p
	1<=j <=4
i:	p+1<=i <= 4+p
	1<=i <=6
p:	-3<= p <= 5

j = i - p - 1

1 <= i - p - 1 <= 4

p + 1 + 1 <= i <= 4 + p + 1

p + 2 <= i <= p + 5

Make Loop Bounds Tighter

S1

j: i-p-1<= j <= i-p-1
1<=j <= 4
i: p+2<=i <= p+5
1<=i <= 6
p: -4<= p<= 4

S2

j: i-p<=j <= i-p
1<=j <=4
i: p+1<=i <= 4+p
1<=i <=6
p: -3<= p <= 5

Union result:

j: i-p-1<=j <= i-p
1<=j <=4
i: p+1<=i <= 5+p
1<=i <=6
p: -4<= p <= 5

Make Loop Bounds Tighter

```
forall (p=-4; p<=5; p++)
   for (i=1; i<=6; i++)
      for (j=1; j<=4; j++){
         if (p== i-j-1)
            X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
         if (p== i-j)
            Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
      }
```

Union result:

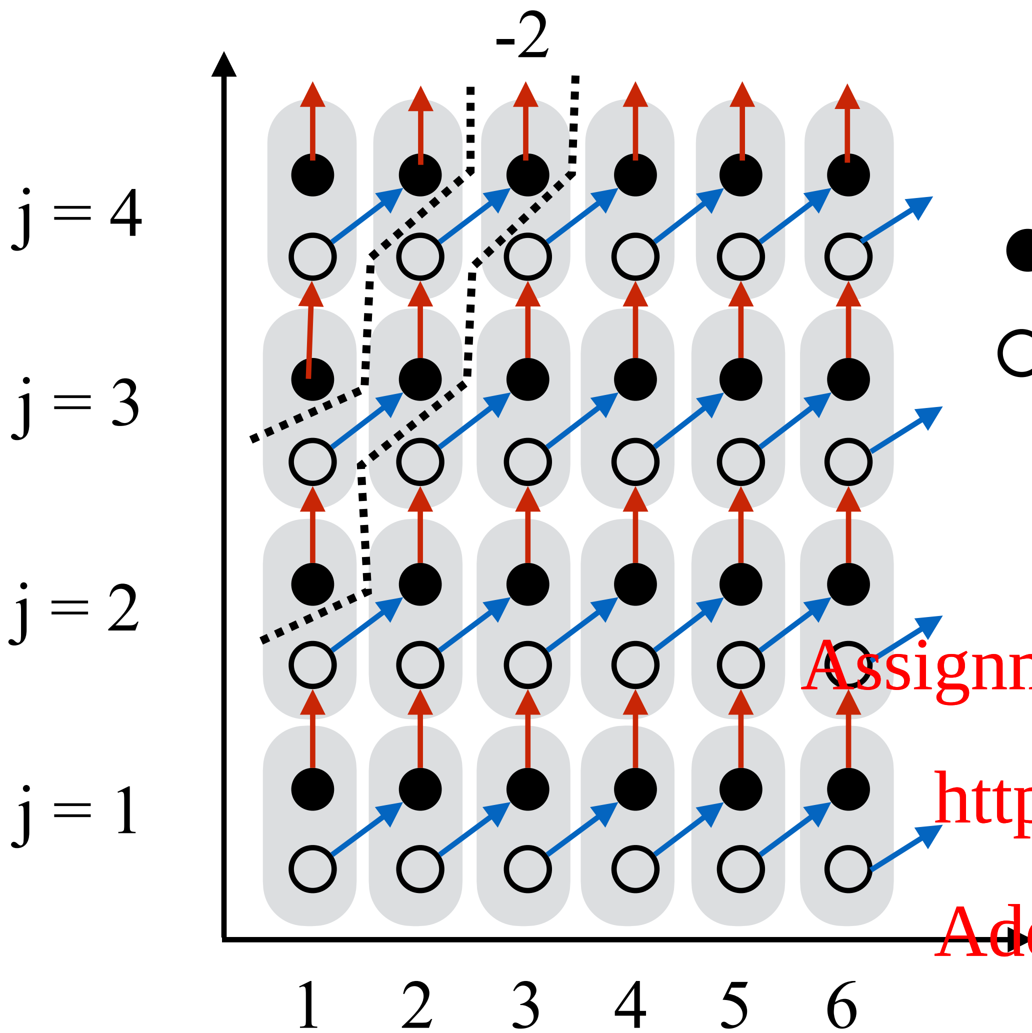
j: i-p-1<=j <= i-p
 1<=j <=4

i: **p+1**<=i <= 5+p
 1<=i <=6

p: -4<= p <= 5


```
for (p=-4; p<=5; p++)
   for (i=max(1,p+1); i<=min(6,5+p); i++)
      for (j=max(1,i-p-1); j<=min(4,i-p); j++){
         if (p== i-j-1)
            X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
         if (p== i-j)
            Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
      }
```

Make Loop Bounds Tighter



When p = -2,

i: [1, 3]

j: [i+1, min(4, i+2)]

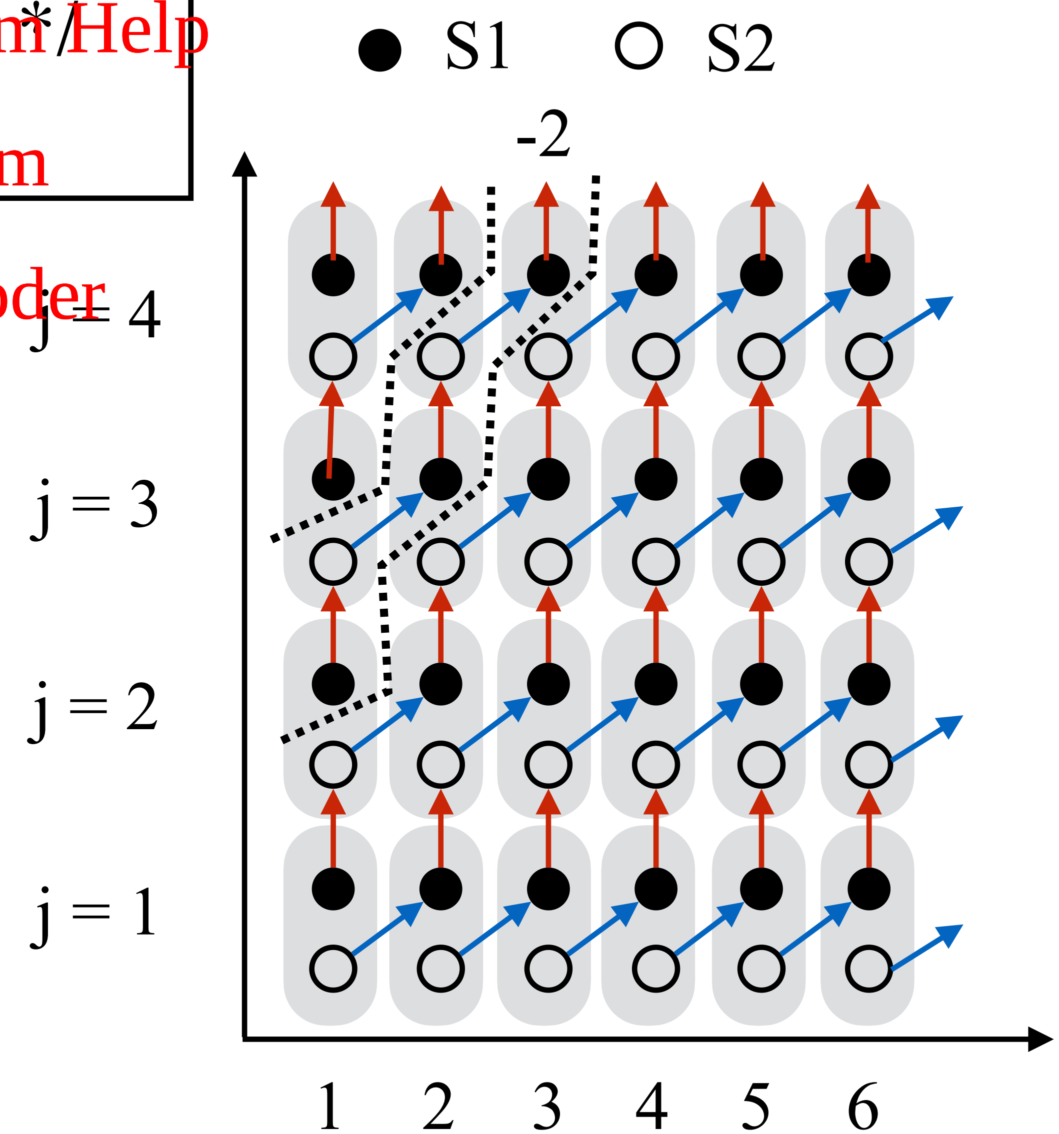

```
for (p=-4; p<=5; p++)
    for (i=max(1,p+1); i<=min(6,5+p); i++)
        for (j=max(1,i-p-1); j<=min(4, i-p); j++)
        {
            if (p== i-j-1)
                X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
            if (p== i-j)
                Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
        }
```

Remove Tests

```
for (p=-4; p<=5; p++)
   for (i=max(1,p+1); i<=min(6,5+p); i++)
      for (j=max(1,i-p-1); j<=min(4,i-p); j++){
         if (p== i-j-1)
            X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
         if (p== i-j)
            Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
      }
```


~~↓?~~

```
for (p=-4; p<=5; p++)
   for (i=max(1,p+1); i<=min(6,5+p); i++)
      j=i-p-1;
      X[i,j] = X[i,j] + Y[i-1, j];    /* S1 */
      j=i-p;
      Y[i,j] = Y[i,j] + X[i, j-1];    /* S2 */
   }
```



Next Class

Reading

- ALSU, Chapter 11.1 - 11.7