# Principle of Polyhedral model – for loop optimization

cschen 陳鍾樞

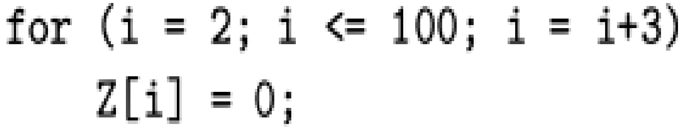
# Outline

- Abstract model
  - Affine expression, Polygon space → Polyhedron space, Affine Accesses
- Data reuse → Data locality
- Tiling
- Space partition
  - Formulate include:
    - Iteration (Variable) space: loop index i, j, …
    - Data dependence
    - Processor mapping
    - Code generation
    - Primitive affine transforms
- Synchronization Between Parallel Loops
- Pipelining
  - Formulate
- Other uses of affine transforms
  - Machine model & features

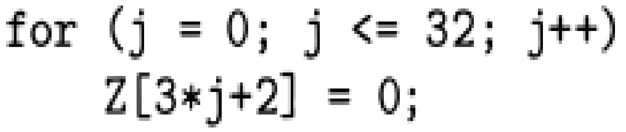
# Affine expression

- $c_0 + c_1 v_1 + \cdots + c_n v_n$, where $c_0, c_1, \cdots, c_n$ are constants.
- Such expressions are informally known as linear expressions.
- Strictly speaking, an affine expression is linear only if $c_0$ is zero.

# Iteration Spaces – Construct



```
for (i = 2; i <= 100; i = i+3)
    Z[i] = 0;
```



```
for (j = 0; j <= 32; j++)
    Z[3*j+2] = 0;
```

# Polygon space

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

Figure 11.10: A 2-dimensional loop nest

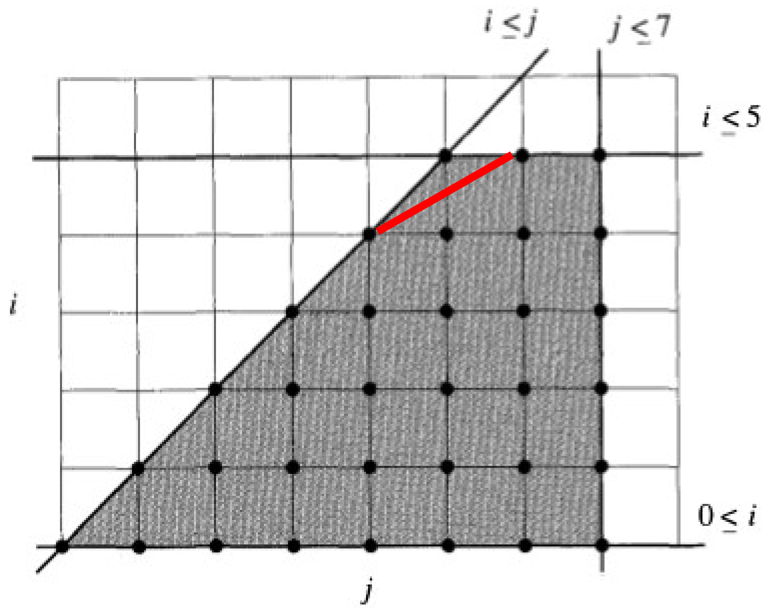
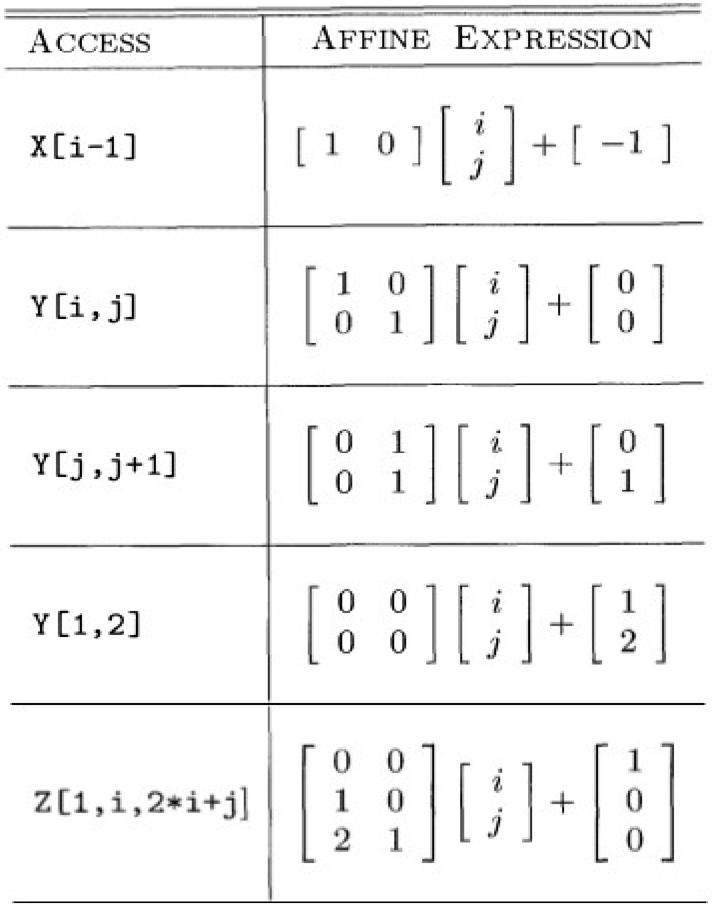


Figure 11.11: The iteration space of Example 11.6

$$\{\mathbf{i} \text{ in } Z^d \mid \mathbf{B}\mathbf{i} + \mathbf{b} \ge \mathbf{0}\}$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

# Affine Accesses



| ACCESS | AFFINE EXPRESSION |
|---|---|
| `X[i-1]` | $\begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} -1 \end{bmatrix}$ |
| `Y[i,j]` | $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ |
| `Y[j,j+1]` | $\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ |
| `Y[1,2]` | $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ |
| `Z[1,i,2*i+j]` | $\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ |

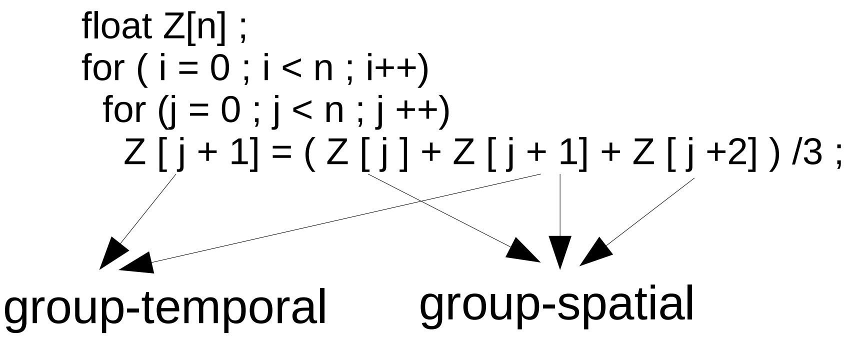
Figure 11.18: Some array accesses and their matrix-vector representations

$A[i^2, i*j]$: is not linear and cannot be solved by Polyhedral.

# Data Reuse

- Self vs Group
  - Self: from the same access
  - Group: from different accesses
- Temporal vs Spatial
  - Temporal: same exact location
  - Spatial: same cache line

# Data Reuse



```
float Z[n] ;
for ( i = 0 ; i < n ; i++)
  for (j = 0 ; j < n ; j ++)
    Z [ j + 1] = ( Z [ j ] + Z [ j + 1] + Z [ j +2] ) /3 ;
```

(0,0),(1,0) Z[ j ]: self-temporal
(0,0),(0,1) Z[ j ]: self-spatial

*opportunities* :

$4n^2$ *accesses* → bring in about $n/c$ cache lines into the cache,
where c is the cache line size

$n$ : self-temporal reuse, $c$ : self-spatial locality,
4 : group reuse

# Data reuse – Null space

$Let\, i\, \text{and}\, i'\, \text{be two iterations that refer to the same array element.}$

$Fi+f=Fi'+f$ ⇒ $F(i-i')=0$

The set of all solutions to the equation Fv = 0 is called the null space of F.

Thus, two iterations refer to the same array element if the difference of their loop-index vectors belongs to the null space of matrix F.

```
for (i = 0; i <= m; i++)
    for (j = 1; j <= n; j++) {
        ...Y[ j, j+1];
    }
```

⇒

$$F_1\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 0 \\ 1 \end{bmatrix}=F_2\begin{bmatrix} i' \\ j' \end{bmatrix}+\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\rightarrow\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 0 \\ 1 \end{bmatrix}=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i' \\ j' \end{bmatrix}+\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\rightarrow\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i-i' \\ j-j' \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\rightarrow \text{let } \begin{bmatrix} i \\ j \end{bmatrix}=\begin{bmatrix} 1 \\ 1 \end{bmatrix}\wedge\begin{bmatrix} i' \\ j' \end{bmatrix}=\begin{bmatrix} 2 \\ 1 \end{bmatrix},\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1-2 \\ 1-1 \end{bmatrix}=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} -1 \\ 0 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Also, the null space is truly a vector space.

$That\, is, if\; Fv_1=0\wedge Fv_2=0, then\; F(v_1+v_2)=0\wedge F(cv_1)=0.$

$$\text{let } v_1=\begin{bmatrix} -1 \\ 0 \end{bmatrix}\wedge v_2=\begin{bmatrix} 3 \\ 0 \end{bmatrix}, Fv_1=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} -1 \\ 0 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}\wedge Fv_2=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ 0 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\rightarrow F(v_1+v_2)=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\left(\begin{bmatrix} -1 \\ 0 \end{bmatrix}+\begin{bmatrix} 3 \\ 0 \end{bmatrix}\right)=\begin{bmatrix} 0 \\ 0 \end{bmatrix}, F(cv_1)=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\left(c\begin{bmatrix} -1 \\ 0 \end{bmatrix}\right)=\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} -c \\ 0 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

# Data reuse – Null space

```
for (i = 0; i <= m; i++)
    for (j = 1; j <= n; j++)
        ...Y[1, 2];
```

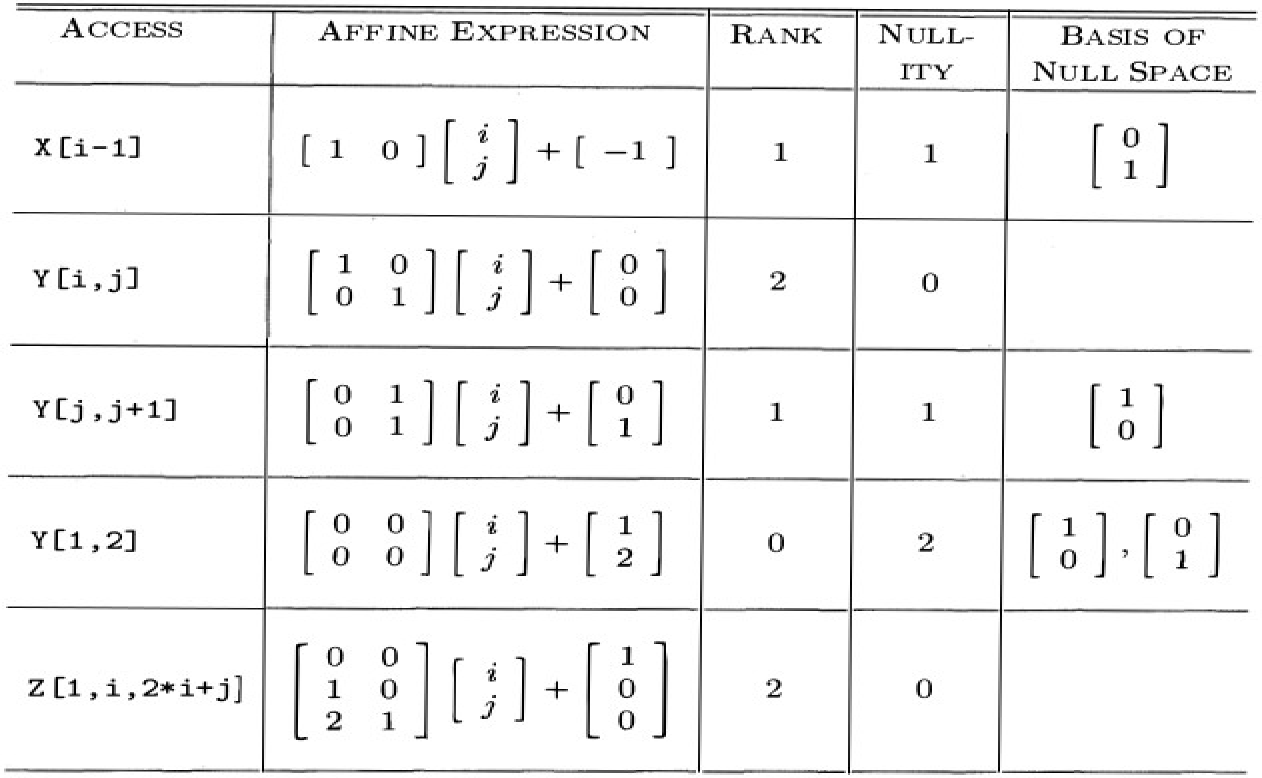
$$\begin{bmatrix}0&0\\0&0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}1\\2\end{bmatrix}=\begin{bmatrix}0&0\\0&0\end{bmatrix}\begin{bmatrix}i'\\j'\end{bmatrix}+\begin{bmatrix}1\\2\end{bmatrix}$$

$$\rightarrow\begin{bmatrix}0&0\\0&0\end{bmatrix}\begin{bmatrix}i-i'\\j-j'\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$

```
for (i = 0; i <= m; i++)
    for (j = 1; j <= n; j++)
        ...Y[ i, j ];
```

$$\rightarrow\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}=\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}i'\\j'\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

$$\rightarrow\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}i-i'\\j-j'\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$



| ACCESS | AFFINE EXPRESSION | RANK | NULLITY | BASIS OF NULL SPACE |
|---|---|---|---|---|
| `X[i-1]` | $\begin{bmatrix}1&0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}-1\end{bmatrix}$ | 1 | 1 | $\begin{bmatrix}0\\1\end{bmatrix}$ |
| `Y[i,j]` | $\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$ | 2 | 0 | |
| `Y[j,j+1]` | $\begin{bmatrix}0&1\\0&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\1\end{bmatrix}$ | 1 | 1 | $\begin{bmatrix}1\\0\end{bmatrix}$ |
| `Y[1,2]` | $\begin{bmatrix}0&0\\0&0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}1\\2\end{bmatrix}$ | 0 | 2 | $\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$ |
| `Z[1,i,2*i+j]` | $\begin{bmatrix}0&0\\1&0\\2&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}1\\0\\0\end{bmatrix}$ | 2 | 0 | |

All j iterations accesses to the same array

Figure 11.19: Rank and nullity of affine accesses

| | | | | |
|---|---|---|---|---|
| $Y[i+j+1,2i+2j]$ | $\begin{bmatrix}1&1\\2&2\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}1\\0\end{bmatrix}$ | 1 | 1 | $\begin{bmatrix}1\\-1\end{bmatrix}$ |

# Data Reuse –
# Self-spatial Reuse

```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
         X[i, j] += Y[1, i, 2*i+j];
```

$$\begin{bmatrix}0 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}$$

$$\rightarrow \begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix}$$

take array elements to share a cache line if they differ only in the last dimension.

$$\rightarrow \begin{bmatrix}0\\ 0\end{bmatrix}$$

In null space

```
for (j = 0; j <=n; j++)
    for (i = 0; i <= m; i++)
         X[i, j] += Y[1, i, 2*i+j];
```

$$\begin{bmatrix}0 & 0\\ 0 & 1\\ 1 & 2\end{bmatrix}\begin{bmatrix}j\\ i\end{bmatrix}$$

$$\rightarrow \begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix}$$

$$\rightarrow \begin{bmatrix}0\\ 1\end{bmatrix}$$

Not in null space

# Data Reuse –
# Self-Temporal Reuse

```
for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++) {
            Out(1, i, j, A[i, j, i+j]);
            Out(2, i, j, A[i+1, j, i+j]);
        }
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

→One solution for vector $v = \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \\ k_1 - k_2 \end{bmatrix}$ is $v = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$; that is, $i_1 = i_2 + 1,\ j_1 = j_2$, and $k_1 = k_2$

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}, Null\, basis\, vector \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

→$k$ always put in inner most and consider other reuse, $i_1 = i_2 + 1,\ j_1 = j_2$

# Data Reuse –
# Self-Temporal Reuse

```
for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++) {
            Out(1, i, j, A[i, j, i+j]);
            Out(2, i, j, A[i+1, j, i+j]);
        }
```

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        for (k = 1; k <= n; k++) {
            Out(1, i, j, A[i, j, i+j]);
            Out(2, i, j, A[i+1, j, i+j]);
        }
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$\rightarrow$ One solution for vector $v = \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \\ k_1 - k_2 \end{bmatrix}$ is $v = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$; that is, $i_1 = i_2 + 1$, $j_1 = j_2$, and $k_1 = k_2$

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}, \mathit{Null\ basis\ vector} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$\rightarrow$ $k$ always put in inner most and consider other reuse, $i_1 = i_2 + 1$, $j_1 = j_2$

Selftemporal reuse gives the most benefit:
a reference with a m-dimensional null space reuses the same data $O(n^m)$ times.

# Data Reuse –
# Self-Temporal Reuse

```
for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++) {
            Out(1, i, j, A[i, j, i+j]);
            Out(2, i, j, A[i+1, j, i+j]);
        }
```

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        for (k = 1; k <= n; k++) {
            Out(1, i, j, A[i, j, i+j]);
            Out(2, i, j, A[i+1, j, i+j]);
        }
```

$$\begin{bmatrix}1&0&0\\0&1&0\\1&1&0\end{bmatrix}\begin{bmatrix}i_1\\j_1\\k_1\end{bmatrix}+\begin{bmatrix}0\\0\\0\end{bmatrix}=\begin{bmatrix}1&0&0\\0&1&0\\1&1&0\end{bmatrix}\begin{bmatrix}i_2\\j_2\\k_2\end{bmatrix}+\begin{bmatrix}1\\0\\0\end{bmatrix}$$

→One solution for vector $v=\begin{bmatrix}i_1-i_2\\j_1-j_2\\k_1-k_2\end{bmatrix}$ is $v=\begin{bmatrix}1\\0\\0\end{bmatrix}$; that is, $i_1=i_2+1$, $j_1=j_2$, and $k_1=k_2$

$F=\begin{bmatrix}1&0&0\\0&1&0\\1&1&0\end{bmatrix}$, *Null basis vector* $\begin{bmatrix}0\\0\\1\end{bmatrix}$

→$k$ always put in inner most and consider other reuse, $i_1=i_2+1$, $j_1=j_2$

Selftemporal reuse gives the most benefit:
a reference with a m-dimensional null space reuses the same data $O(n^m)$ times.

# Data Reuse – Group Reuse

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        for (k = 1; k <= n; k++) {
            Out(1, i, j, A[i, j, i+j ]);
            Out(2, i, j, A[i+1, j, i+j ]);
        }
```

```
A[1..n, 1..n, i+j];
A[2..n+1, 1..n, i+j];
```

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}, Null\ basis\ vector \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$\rightarrow k$ always put in inner most and consider other reuse, $i_1 = i_2 + 1,\ j_1 = j_2$

```
for (j = 1; j <= n; j++)
    for (k = 1; k <= n; k++)
        Out(1, 1, j, A[1, j, 1+j ]);
for (i = 2; i <= n; i++) {
    for (j = 1; j <= n; j++)
        for (k = 1; k <= n; k++) {
            Out(1, i, j, A[i, j, i+j ]);
            Out(2, i-1, j, A[i, j, i+j ]);
        }
for (j = 1; j <= n; j++)
    for (k = 1; k <= n; k++)
        Out(2, n, j, A[n+1, j, n+j ]);
```

```
A[1, 1..n, i+j];
A[2..n, 1..n, i+j];
A[2..n, 1..n, i+j];
A[n+1, j, i+j];
```

# Data Reuse – Self Reuse

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++) {
        if (i == j)
            ...A[i, j];
        else
            ...B[ j ];
```

⇓

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        If (i == j)
            ...A[i, j];
```

⇒ Self-Spatial reuse

```
for (j = 1; j <= n; j++)
    for (i = 1; i <= n; i++) {
        If (i != j)
            ...B[ j ];
```

⇒ Self-Temporal reuse

# Data Reuse –
# Data dependence

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++) {
        if (i == j)
            a += A[i, j];
        else {
            b |= B[ j ];
            c += C[ j ];
        }
```

↓

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        If (i == j)
            a += A[i, j];
```

→ Self-Spatial reuse

```
for (j = 1; j <= n; j++)
    for (i = 1; i <= n; i++) {
        If (i != j)
            b |= B[ j ];
```

→ Self-Temporal reuse

```
for (j = 1; j <= n; j++)
    for (i = 1; i <= n; i++) {
        If (i != j)
            c += C[ j ];
```

✗ → Overflow or not.

# Data Reuse –
# Data dependence

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        if (i == j)
            ...
        else {
            C[ i ] += A[ j, i ];
            B[ i ] -= A[ j, i ];
        }
    c += C[ i ];
    b += B[ i ];
}
```

→

```
for (j = 1; j <= n; j++) {
    for (i = 1; i <= n; i++)
        If (i != j)
            C[ j ] += A[ j, i ];
    c += C[ j ];
```

✓ → Ignore Overflow

```
for (j = 1; j <= n; j++)
    for (i = 1; i <= n; i++) {
        If (i != j)
            B[ j ] -= A[ j, i ];
    b -= B[ j ];
```

✗ → Ignore Overflow

For operators with commutativity and associativity, like + or *, it work.
But without commutativity and associativity, like – or /, it not work.

$$(a_0 - a_1) - a_2 \neq a_0 - (a_1 - a_2)$$

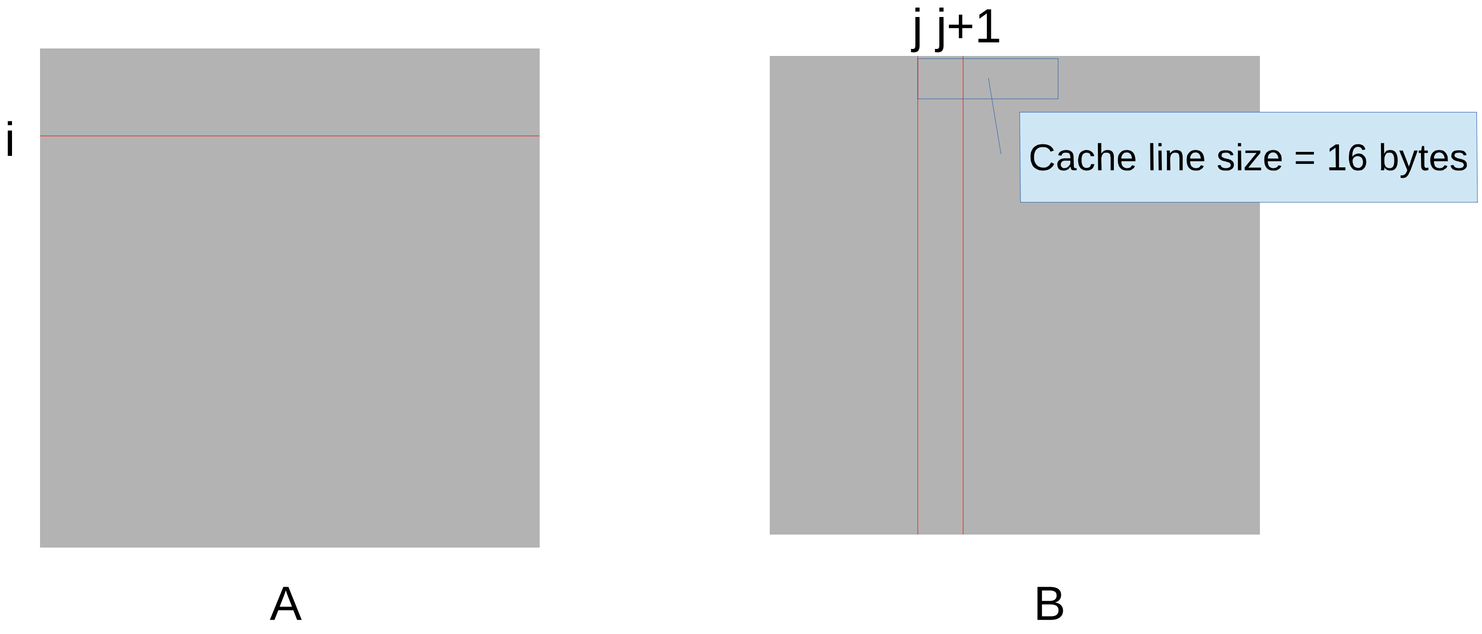
Anyway it is far too complex for compiler.

# Tiling

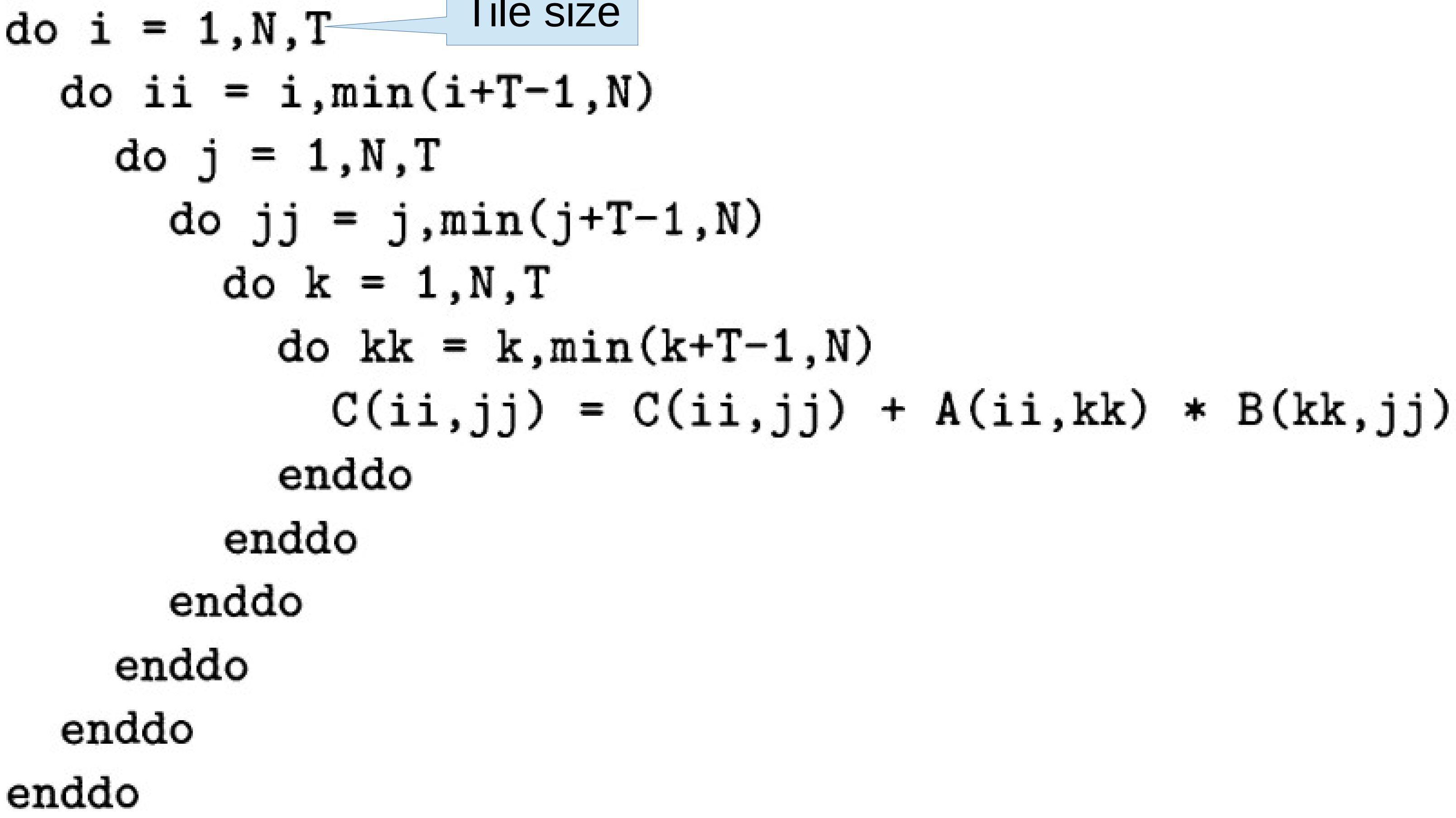
# Data cache miss – regarding B

```
do i = 1,N
  do j = 1,N
    do k = 1,N
      C(i,j) = C(i,j)
         + A(i,k) * B(k,j)
    enddo
  enddo
enddo
```

Assume data cache line size is 4 words,
After B(-,j) is read, the B(-,j+1) is in cache too.
When N is large, the A(i,-)*B(-,j+1) will be cache miss.
B(i,j) B(i,j+1): spatial locality.



# Tiling

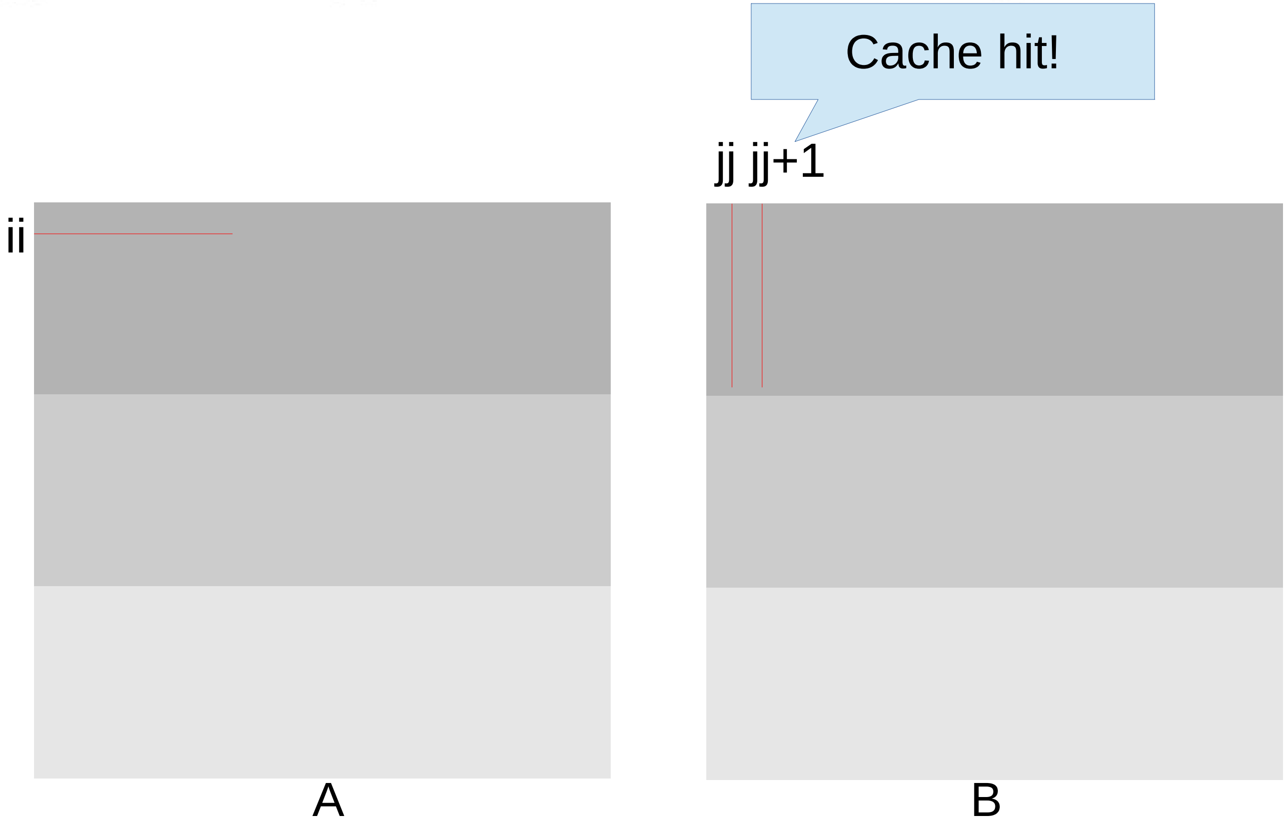


```
do i = 1,N,T
  do ii = i,min(i+T-1,N)
    do j = 1,N,T
      do jj = j,min(j+T-1,N)
        do k = 1,N,T
          do kk = k,min(k+T-1,N)
            C(ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

Tile size

# Tiling - Reduce data cache miss – Regarding B

```
do i = 1,N,T
  do j = 1,N,T
    do k = 1,N,T
      do ii = i,min(i+T-1,N)
        do jj = j,min(j+T-1,N)
          do kk = k,min(k+T-1,N)
            C(ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```



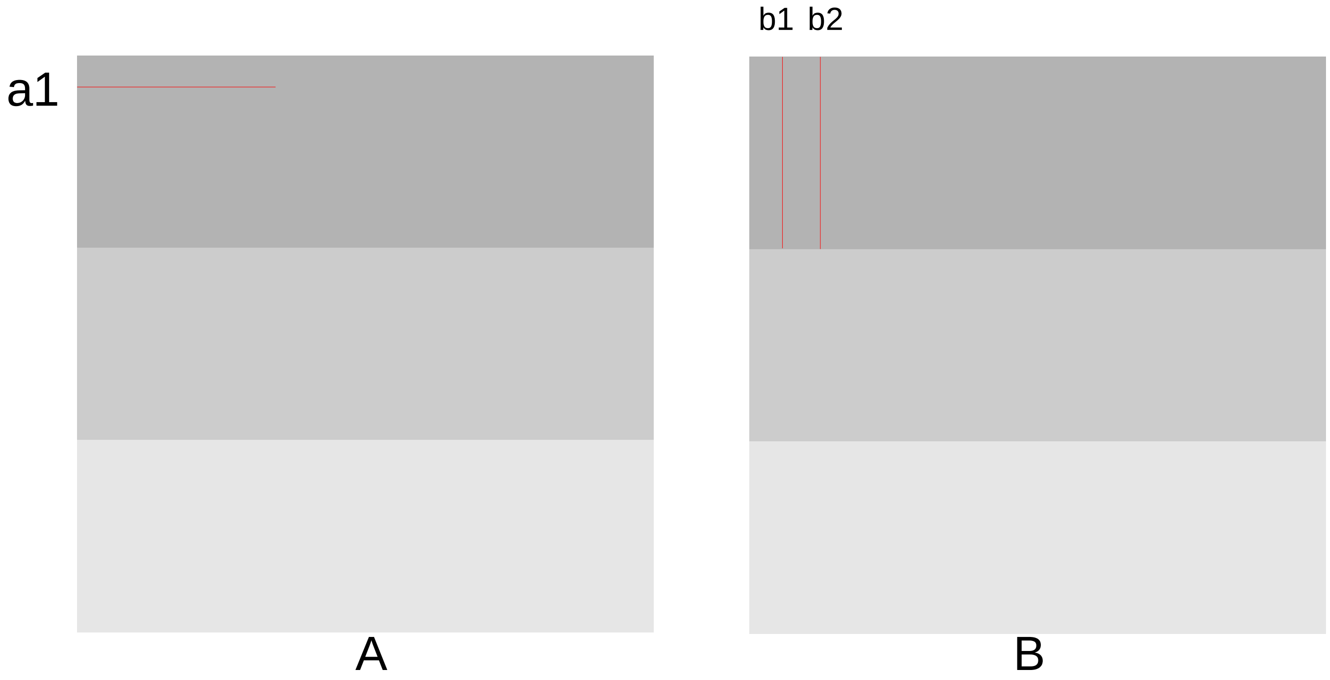
# Tiling - Reduce data cache miss – Regarding A

```
do i = 1,N,T
   do j = 1,N,T
      do k = 1,N,T
         do ii = i,min(i+T-1,N)
            do jj = j,min(j+T-1,N)
               do kk = k,min(k+T-1,N)
                  C(ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

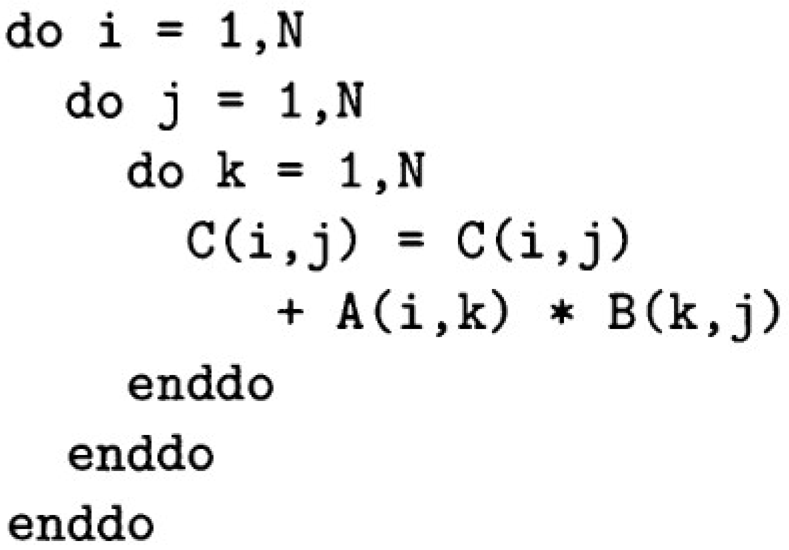
1. a1*b1
2. a1*b2 cache hit
3. a1*b3 hit

…

a1 hit T-1 times, miss 1 time



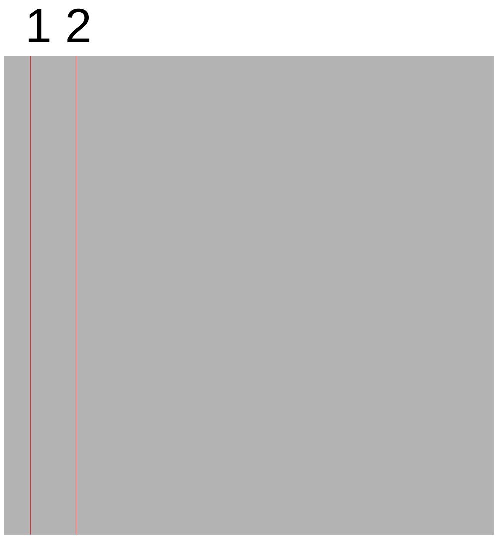
# Data cache miss – Regarding A



```
do i = 1,N
  do j = 1,N
    do k = 1,N
      C(i,j) = C(i,j)
          + A(i,k) * B(k,j)
    enddo
  enddo
enddo
```

1. A[1,1]*B[1,1]
…
2. A[1,N/2+1]*B[N/2+1]
 (A[1,N/2+1] push A[1,1] out)
...
3. A[1,1]*B[2,1] (cache miss A[1,1])
…
A[1,1] never hit
Miss temporal locality

A

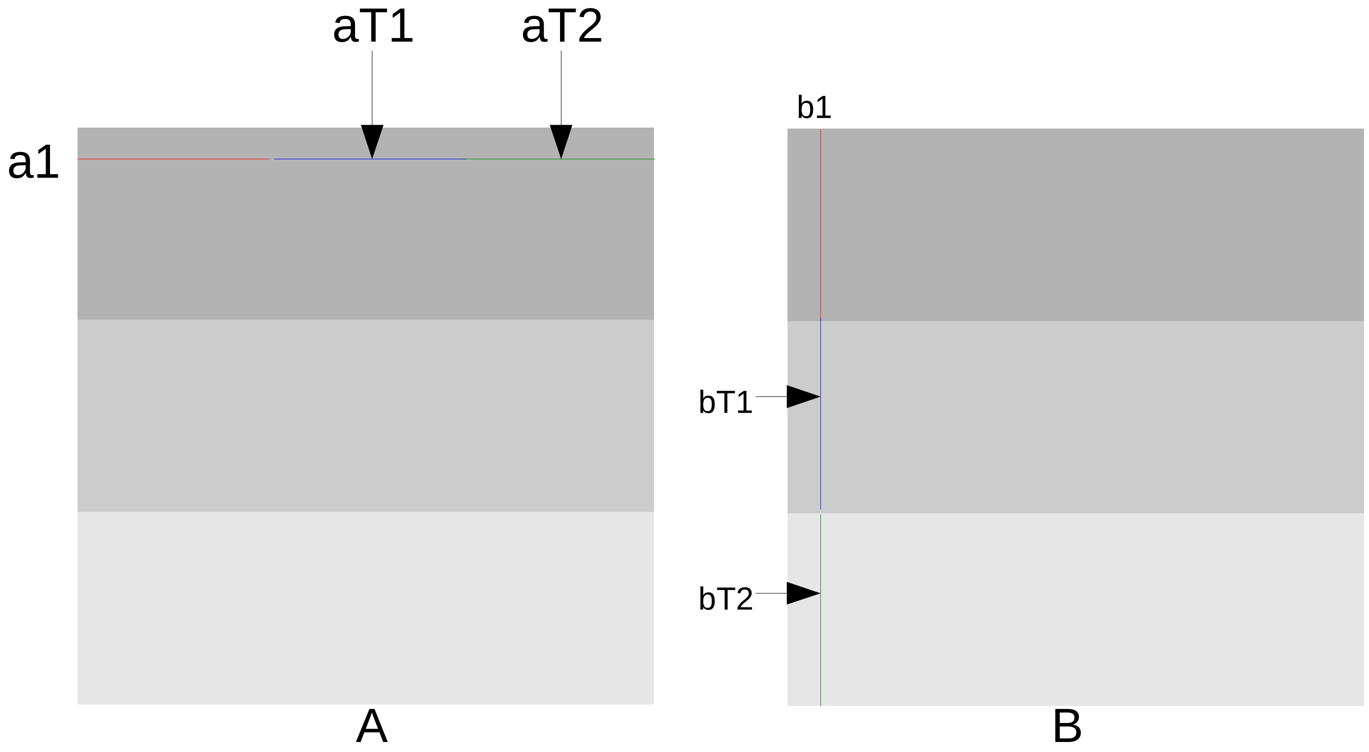


B

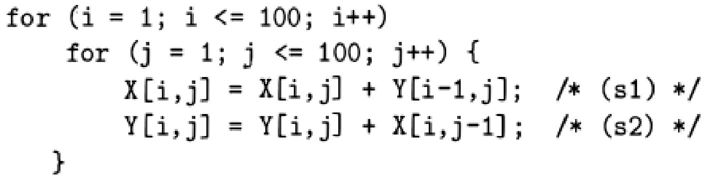
# Tiling correctness – proof

```
do i = 1,N,T
  do j = 1,N,T
    do k = 1,N,T
      do ii = i,min(i+T-1,N)
        do jj = j,min(j+T-1,N)
          do kk = k,min(k+T-1,N)
            C(ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

1. a1*b1 (k=1)
2. aT1*bT1 (k=T+1)
3. aT2*bT2 (k=2T+1)

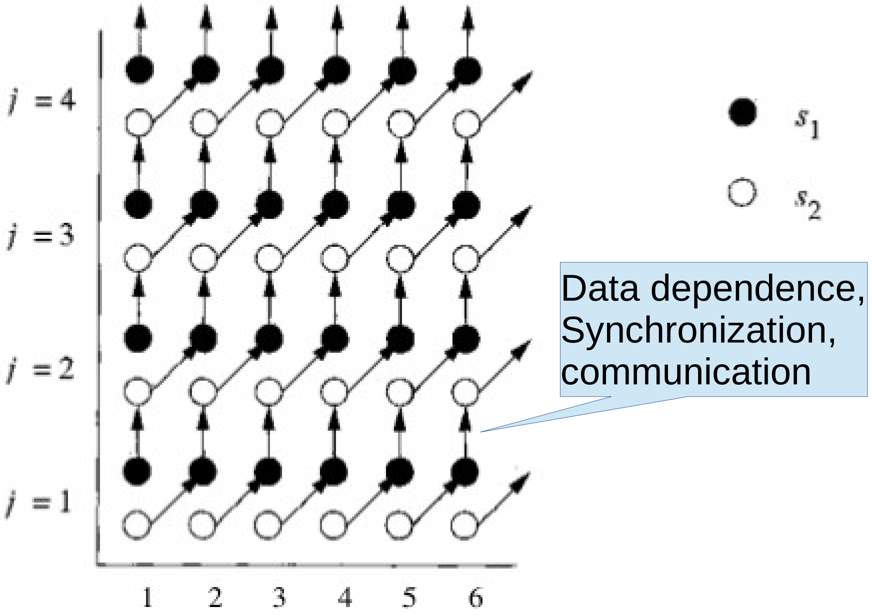


# Loop optimization
# – data locality & parallel



```
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) */
   }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations



Data dependence,
Synchronization,
communication

Figure 11.27: Dependences of the code in Example 11.41

# Processor (partition) mapping

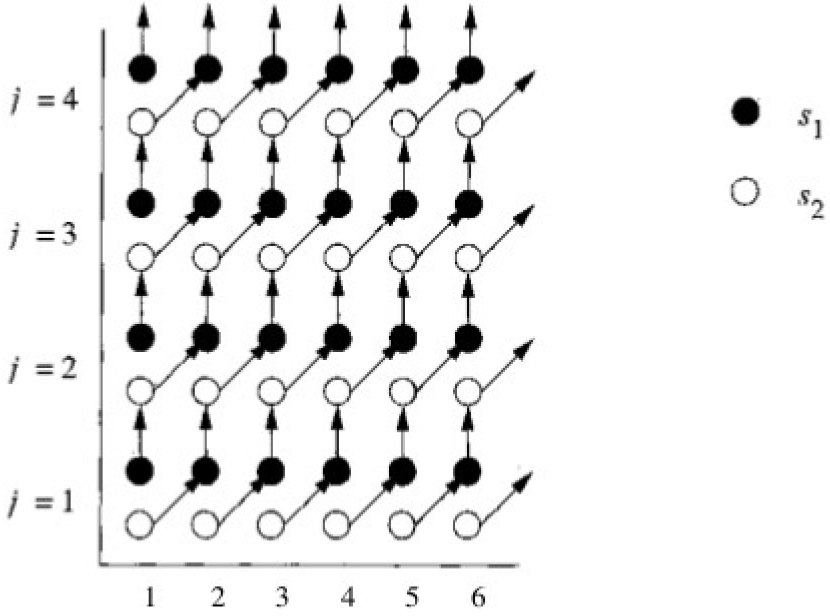


Figure 11.27: Dependences of the code in Example 11.41

$$S_1 : [p] = i - j - 1$$

$$S_2 : [p] = i - j$$

# Processor (partition) mapping

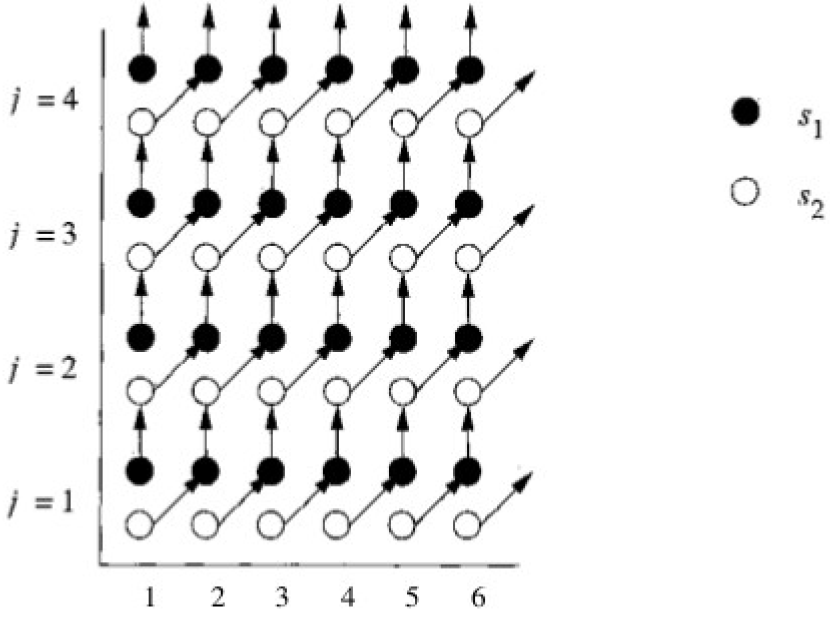


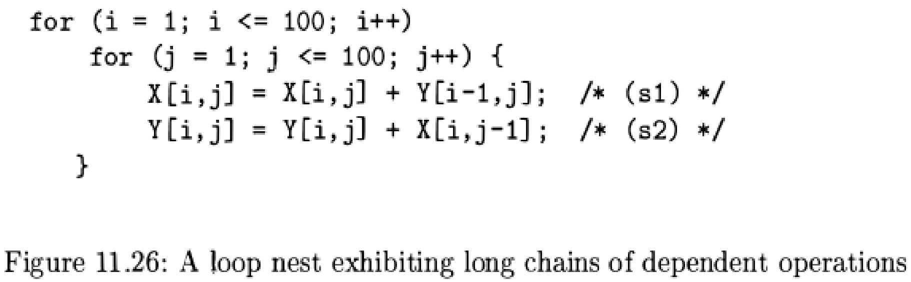
Figure 11.27: Dependences of the code in Example 11.41

$$S_1: [p] = i - j - 1$$

$$S_2: [p] = i - j$$

$s_1: i=1, j=1 \rightarrow p=-1$
$s_2: i=1, j=2 \rightarrow p=-1$
$s_1: i=2, j=2 \rightarrow p=-1$
…

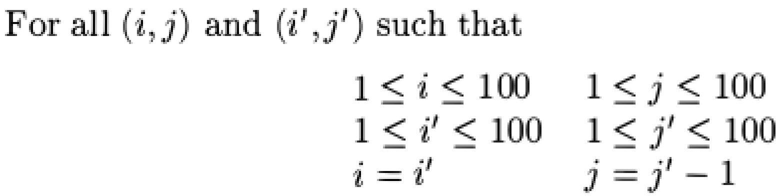
# Processor mapping – get mapping function



```
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) */
   }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations

$j: current, j': next, j = j' - 1 \rightarrow X[i \quad j] = X[i' \quad j' - 1]$



For all $(i, j)$ and $(i', j')$ such that

$$\begin{array}{ll} 1 \le i \le 100 & 1 \le j \le 100 \\ 1 \le i' \le 100 & 1 \le j' \le 100 \\ i = i' & j = j' - 1 \end{array}$$

$[i \quad j], [i' \quad j']$ are assigned to the same processor, otherwise need synchronization.

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + [c_1] = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + [c_2]$$

$$\Rightarrow \begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + [c_1] = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i \\ j+1 \end{bmatrix} + [c_2]$$

$$\Rightarrow (C_{11} - C_{21}) i + (C_{12} - C_{22}) j - C_{22} + c_1 - c_2 = 0$$

$$\Rightarrow \begin{array}{l} C_{11} - C_{21} = 0 \\ C_{12} - C_{22} = 0 \\ c_1 - c_2 - C_{22} = 0 \end{array}$$

$i: current, i': next, i = i' - 1 \rightarrow Y[i \quad j] = Y[i' - 1 \quad j']$

$$\begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + [c_2] = \begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + [c_1]$$

$$\Rightarrow \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + [c_2] = \begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i+1 \\ j \end{bmatrix} + [c_1]$$

$$\Rightarrow (C_{21} - C_{11}) i - C_{11} + (C_{22} - C_{12}) j - c_1 + c_2 = 0$$

$$\Rightarrow \begin{array}{l} C_{11} - C_{21} = 0 \\ C_{12} - C_{22} = 0 \\ c_1 - c_2 + C_{11} = 0 \end{array}$$

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

$$\xrightarrow{pick\, C_{11} = 1} \quad C_{21} = 1, C_{22} = -1, C_{12} = -1$$

$$\xrightarrow{pick\, c_2 = 0} \quad c_1 = -1$$

# Processor mapping – get mapping function

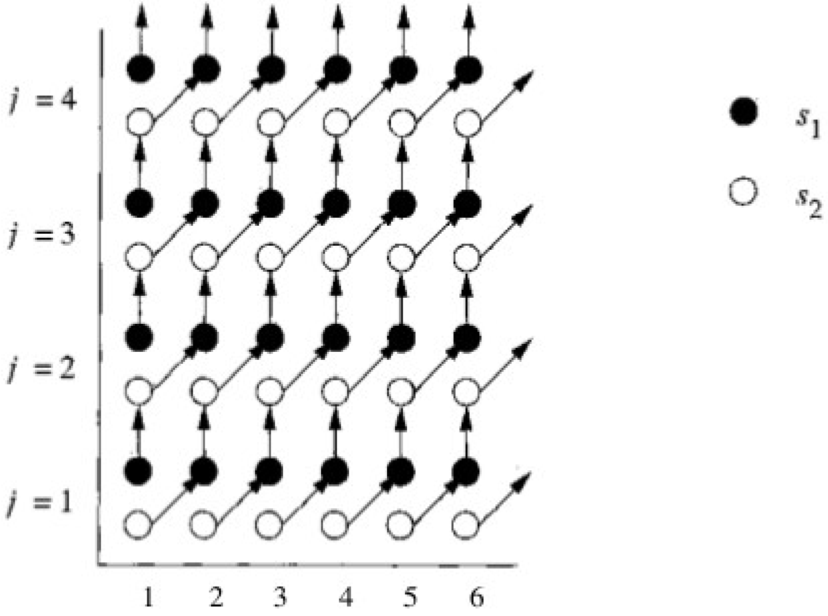


Figure 11.27: Dependences of the code in Example 11.41

$C_{11}=1, C_{21}=1, C_{22}=-1, C_{12}=-1, c_2=0, c_1=-1$

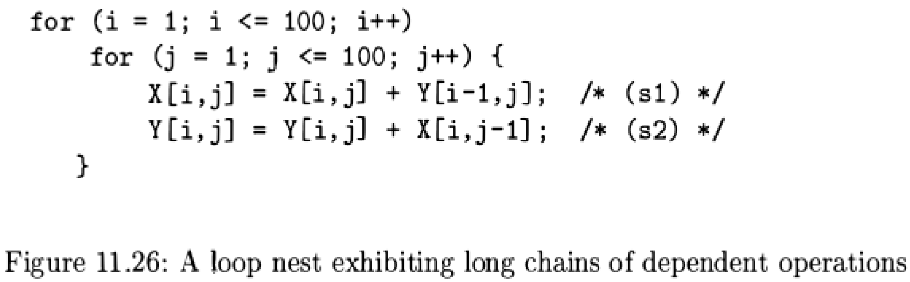
$$s_1:[p]=\begin{bmatrix} C_{11} & C_{12} \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+[c_1]=\begin{bmatrix} 1 & -1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+[-1]=i-j-1$$

$$s_2:[p]=\begin{bmatrix} C_{21} & C_{22} \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+[c_2]=\begin{bmatrix} 1 & -1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+[0]=i-j$$

$S_1:[p]=i-j-1$

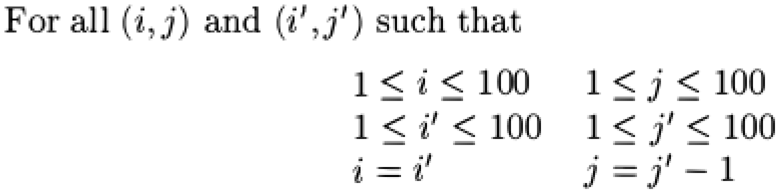
$S_2:[p]=i-j$

# Formulate – variable space & processor mapping



```
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) */
   }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations



For all $(i, j)$ and $(i', j')$ such that

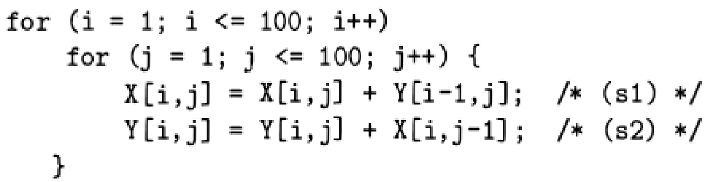
$$\begin{array}{ll} 1 \le i \le 100 & 1 \le j \le 100 \\ 1 \le i' \le 100 & 1 \le j' \le 100 \\ i = i' & j = j' - 1 \end{array}$$

$For\ all\ i_1 \in Z^{d_1}$ and $i_2 \in Z^{d_2}\ such\ that$

1. $B_1 i_1 + b_1 \ge 0$
2. $B_2 i_2 + b_2 \ge 0$
3. $F_1 i_1 + f_1 = F_2 i_2 + f_2$

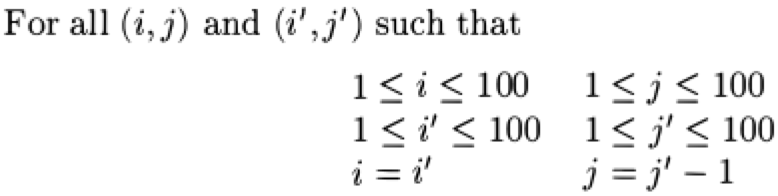
$it\ is\ the\ case\ that\ C_1 i_1 + c_1 = C_2 i_2 + c_2$

# Formulate – variable space & processor mapping



```
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) */
   }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations



For all $(i, j)$ and $(i', j')$ such that

$$\begin{array}{ll} 1 \le i \le 100 & 1 \le j \le 100 \\ 1 \le i' \le 100 & 1 \le j' \le 100 \\ i = i' & j = j' - 1 \end{array}$$

$e.g.\ 1 \le i$ <=> $[1 \quad 0]\begin{bmatrix} i \\ j \end{bmatrix} + [-1] \ge 0$

$For\ all\ i_1 \in Z^{d_1}$ and $i_2 \in Z^{d_2}\ such\ that$

$1.\ B_1 i_1 + b_1 \ge 0$

$2.\ B_2 i_2 + b_2 \ge 0$

$3.\ F_1 i_1 + f_1 = F_2 i_2 + f_2 \quad \rightarrow i = i', j = j' - 1$

$it\ is\ the\ case\ that\ C_1 i_1 + c_1 = C_2 i_2 + c_2$

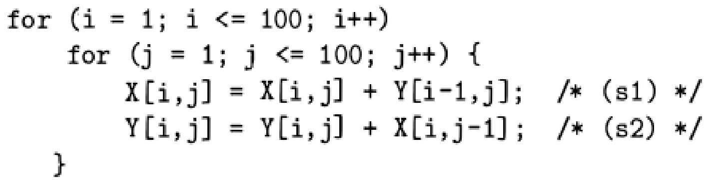
$$[C_{11} \quad C_{12}]\begin{bmatrix} i \\ j \end{bmatrix} + [c_1] = [C_{21} \quad C_{22}]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c_2]$$

$$\rightarrow [1 \quad -1]\begin{bmatrix} i \\ j \end{bmatrix} + [-1] = [1 \quad -1]\begin{bmatrix} i' \\ j' \end{bmatrix} + [0]$$

$$i - j - 1 = i' - j'$$

Convex region in d-dimension space.
It can be reduced to integer-linear-programming which use linear algebra.

# Formulate – data dependence



```
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) */
   }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations

$$For\, all\, i_1 \in Z^{d_1} \text{ and } i_2 \in Z^{d_2}\, such\, that$$

$$F_1 i_1 + f_1 = F_2 i_2 + f_2$$

$$Let\, i_1 = \begin{bmatrix} i \\ j \end{bmatrix}, i_2 = \begin{bmatrix} i' \\ j' \end{bmatrix}$$

$$F_1 i_1 + f_1 = F_2 i_2 + f_2 \Rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\Rightarrow i = i', j = j' - 1$$

*If no solution → no data dependence → full parallel like matrix multiplication.*

# NP-complete problem

$For\, all\, i_1 \in Z^{d_1}$ and $i_2 \in Z^{d_2}\, such\, that$

$1. B_1 i_1 + b_1 \geq 0$
$2. B_2 i_2 + b_2 \geq 0$
$3. F_1 i_1 + f_1 = F_2 i_2 + f_2$

Search for integer solutions that satisfy a set of linear inequalities, which is precisely the well-known problem of integer linear programming.

Integer linear programming is an NP-complete problem.

# Formulate – variable space, data dependence & processor mapping

*For all* $i_1 \in Z^{d_1}$ and $i_2 \in Z^{d_2}$ *such that*

1. $B_1 i_1 + b_1 \geq 0$
2. $B_2 i_2 + b_2 \geq 0$
3. $F_1 i_1 + f_1 = F_2 i_2 + f_2$

*it is the case that* $C_1 i_1 + c_1 = C_2 i_2 + c_2$

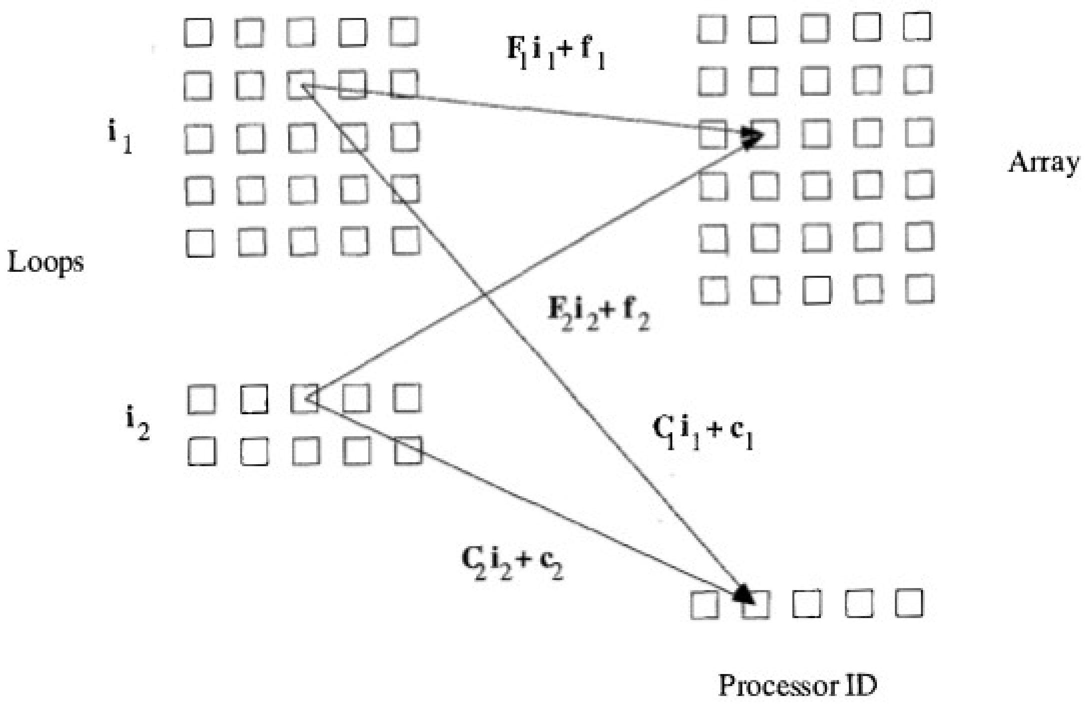


Figure 11.25: Space-partition constraints

# Formulate – variable space dimension

$For\ all\ i_1 \in Z^{d_1}$ and $i_2 \in Z^{d_2}\ such\ that$

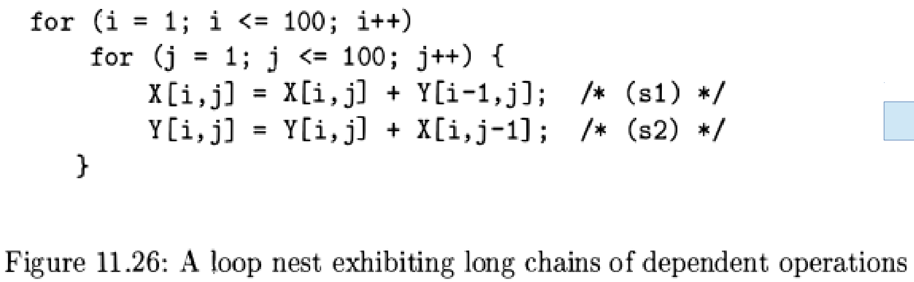
1. $B_1 i_1 + b_1 \geq 0$
2. $B_2 i_2 + b_2 \geq 0$
3. $F_1 i_1 + f_1 = F_2 i_2 - f_2$

$it\ is\ the\ case\ that\ C_1 i_1 + c_1 = C_2 i_2 + c_2$

$$example\ of\ i_1 \in Z^{d_1} \wedge i_2 \in Z^{d_2},\ where\ d_1 = \mathbf{4},\ d_2 = \mathbf{2}$$

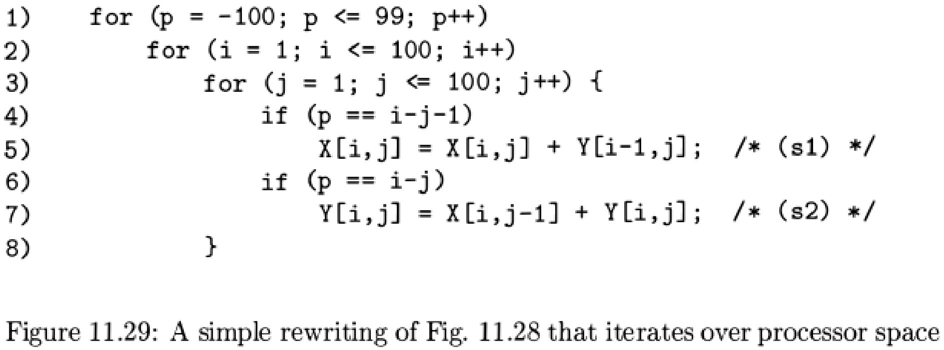
```
int X[100][100][100][100];

for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    for (k=0; k<100; k++)
      for (l=0; l<100; l++)
        X[i][j][k][l] = X[i][j][k+1][l-1];


for (k=0; k<100; k++)
  for (l=0; l<100; l++)
    X[k][l][k][l+1] = X[k][l][k][l];
```

# Code generation



```
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) */
    }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations

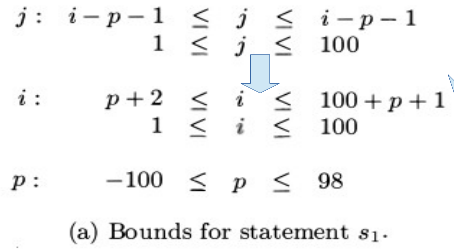


```
for (p = -100; p <= 99; p++)
    for (i = 1; i <= 100; i++)
        for (j = 1; j <= 100; j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j];   /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j];   /* (s2) */
        }
```

Figure 11.29: A simple rewriting of Fig. 11.28 that iterates over processor space

Eliminating Empty Iterations

$S_1:[p]=i-j-1 \rightarrow j=i-p-1$

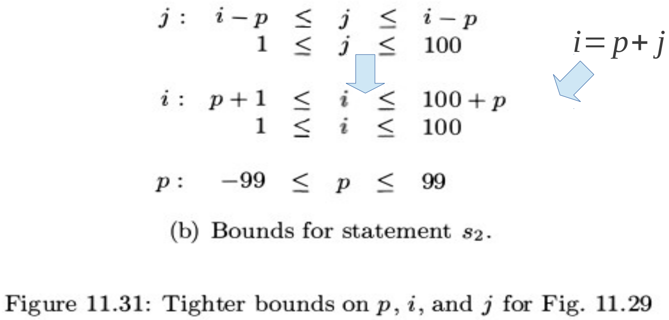


$$\begin{array}{llcccl} j: & i-p-1 & \le & j & \le & i-p-1 \\ & 1 & \le & j & \le & 100 \\ \\ i: & p+2 & \le & i & \le & 100+p+1 \\ & 1 & \le & i & \le & 100 \\ \\ p: & -100 & \le & p & \le & 98 \end{array}$$

$i=p+1+j$

(a) Bounds for statement $s_1$.

$S_2:[p]=i-j \rightarrow j=i-p$

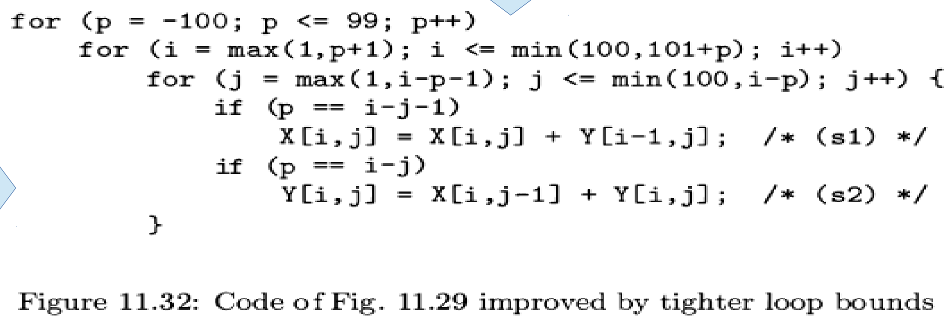


$$\begin{array}{llcccl} j: & i-p & \le & j & \le & i-p \\ & 1 & \le & j & \le & 100 \\ \\ i: & p+1 & \le & i & \le & 100+p \\ & 1 & \le & i & \le & 100 \\ \\ p: & -99 & \le & p & \le & 99 \end{array}$$

$i=p+j$

(b) Bounds for statement $s_2$.

Figure 11.31: Tighter bounds on $p$, $i$, and $j$ for Fig. 11.29



```
for (p = -100; p <= 99; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,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] = X[i,j-1] + Y[i,j];   /* (s2) */
        }
```

Figure 11.32: Code of Fig. 11.29 improved by tighter loop bounds

# Code generation – Eliminating Test from Innermost Loops

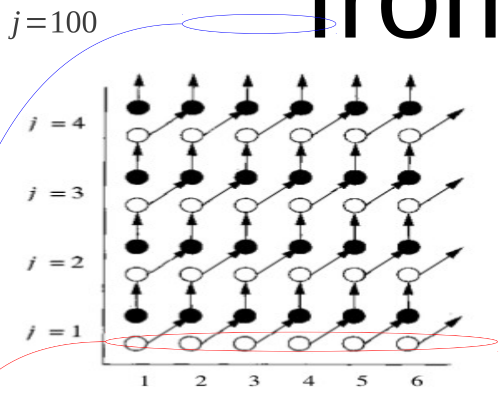
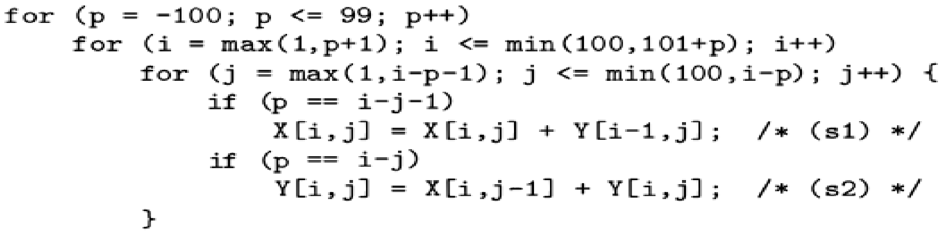
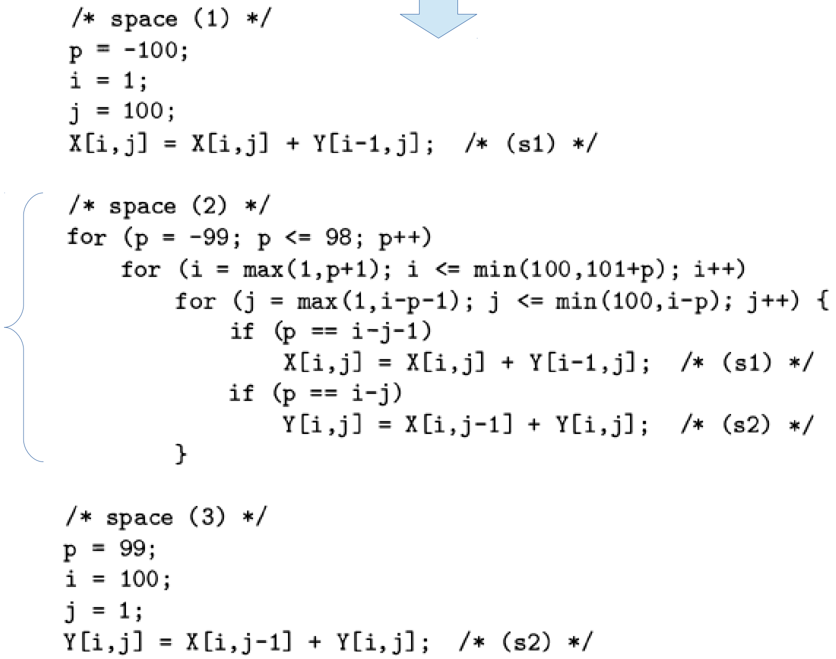


Figure 11.27: Dependences of the code in Example 11.41



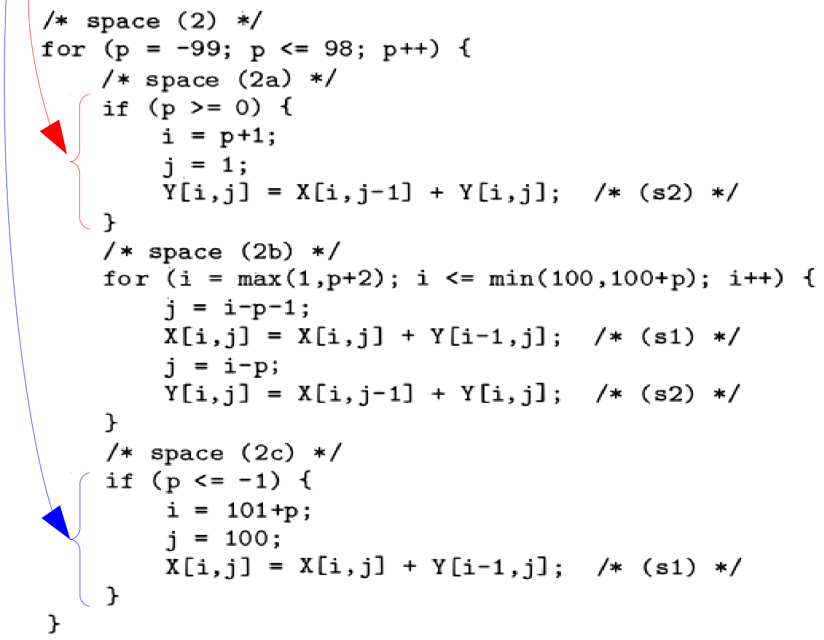
```
for (p = -100; p <= 99; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j];   /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j];   /* (s2) */
        }
```

Figure 11.32: Code of Fig. 11.29 improved by tighter loop bounds



```
/* space (1) */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j];   /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,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] = X[i,j-1] + Y[i,j];   /* (s2) */
        }

/* space (3) */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j];   /* (s2) */
```

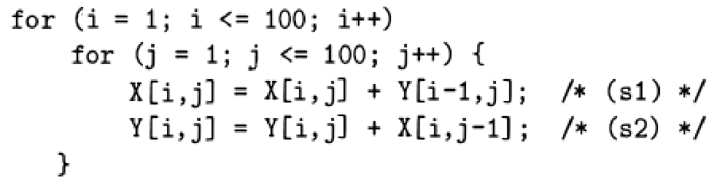
Figure 11.33: Splitting the iteration space on the value of $p$



```
/* space (2) */
for (p = -99; p <= 98; p++) {
    /* space (2a) */
    if (p >= 0) {
        i = p+1;
        j = 1;
        Y[i,j] = X[i,j-1] + Y[i,j];   /* (s2) */
    }
    /* space (2b) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        j = i-p-1;
        X[i,j] = X[i,j] + Y[i-1,j];   /* (s1) */
        j = i-p;
        Y[i,j] = X[i,j-1] + Y[i,j];   /* (s2) */
    }
    /* space (2c) */
    if (p <= -1) {
        i = 101+p;
        j = 100;
        X[i,j] = X[i,j] + Y[i-1,j];   /* (s1) */
    }
}
```

(a) Splitting space (2) on the value of $i$.

# Code generation – Eliminating Test from Innermost Loops

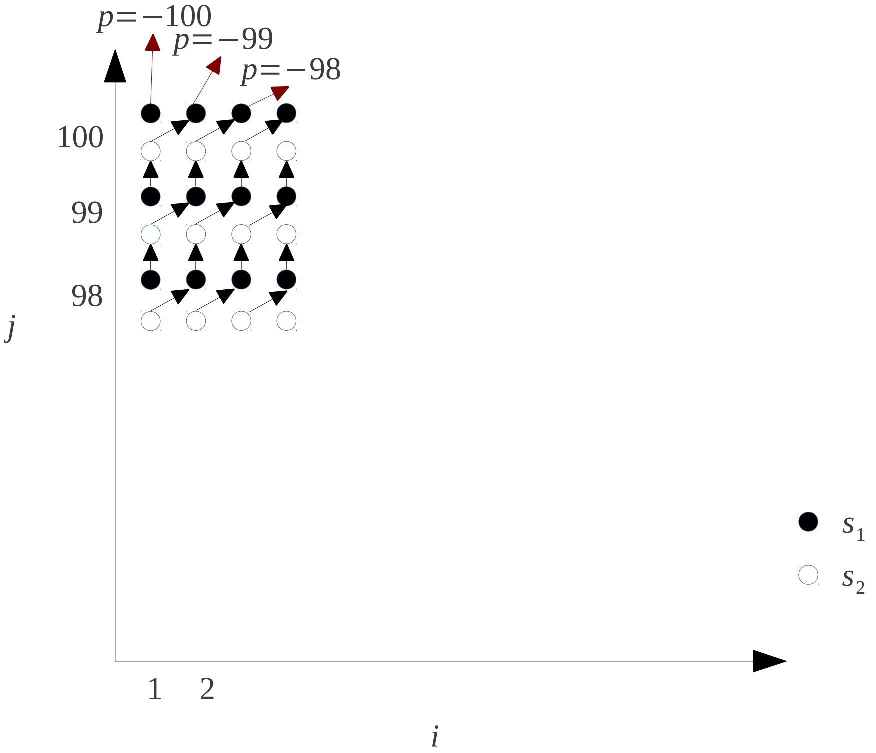


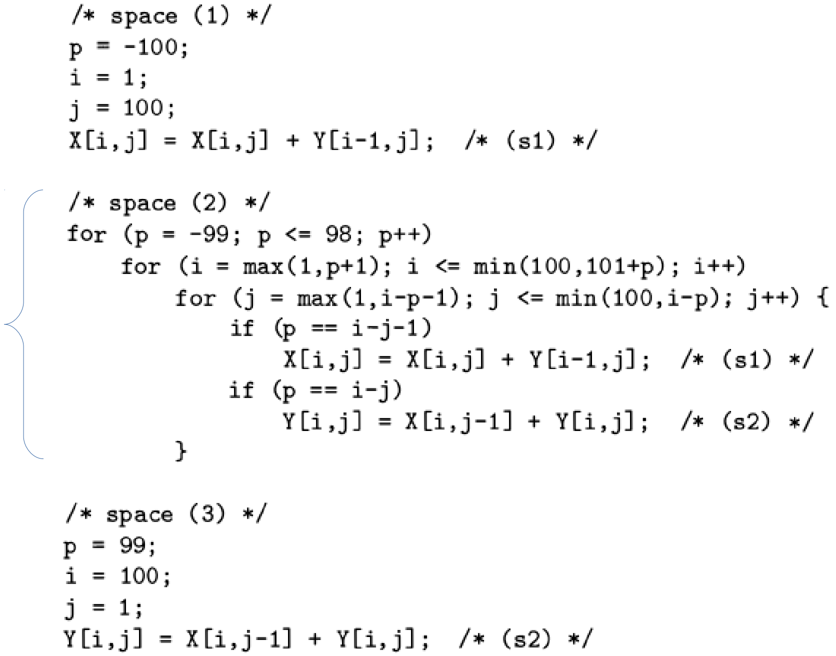
```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1];  /* (s2) */
  }
```

Figure 11.26: A loop nest exhibiting long chains of dependent operations

p=-99, i=max(1, -98)..min(100, 2)=1..2,
i:1, j=max(1, 99)..min(100, 100)=99..100
i:2, j=100..100
p=-98, i=max(1, -97)..min(100, 3)=1..3,
i:1, j=max(1, 98)..min(100, 99)=98..99
i:2, j=99..100
I:3, j=100..100

p=-99, i=1,
j=99 → s1, j=100 → s2
p=-99, i=2,
j=100 → s1

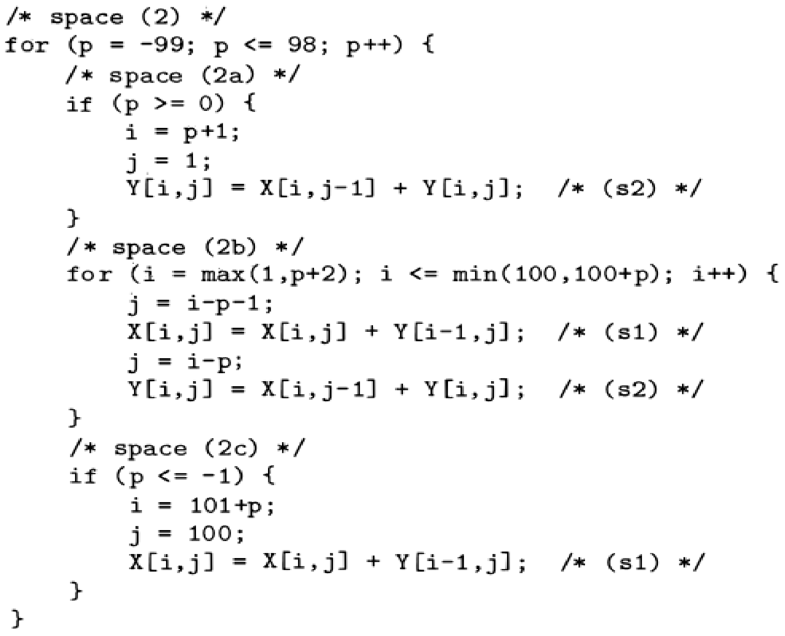




```
/* space (1) */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,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] = X[i,j-1] + Y[i,j];  /* (s2) */
        }

/* space (3) */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
```

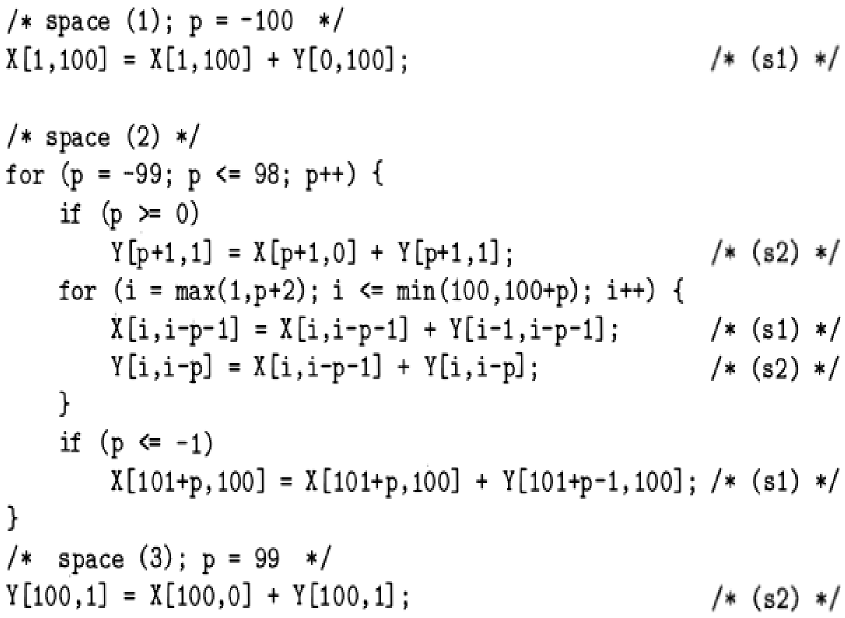
Figure 11.33: Splitting the iteration space on the value of $p$

# Code generation



```
/* space (2) */
for (p = -99; p <= 98; p++) {
    /* space (2a) */
    if (p >= 0) {
        i = p+1;
        j = 1;
        Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
    }
    /* space (2b) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        j = i-p-1;
        X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
        j = i-p;
        Y[i,j] = X[i,j-1] + Y[i,j];  /* (s2) */
    }
    /* space (2c) */
    if (p <= -1) {
        i = 101+p;
        j = 100;
        X[i,j] = X[i,j] + Y[i-1,j];  /* (s1) */
    }
}
```

(a) Splitting space (2) on the value of $i$.

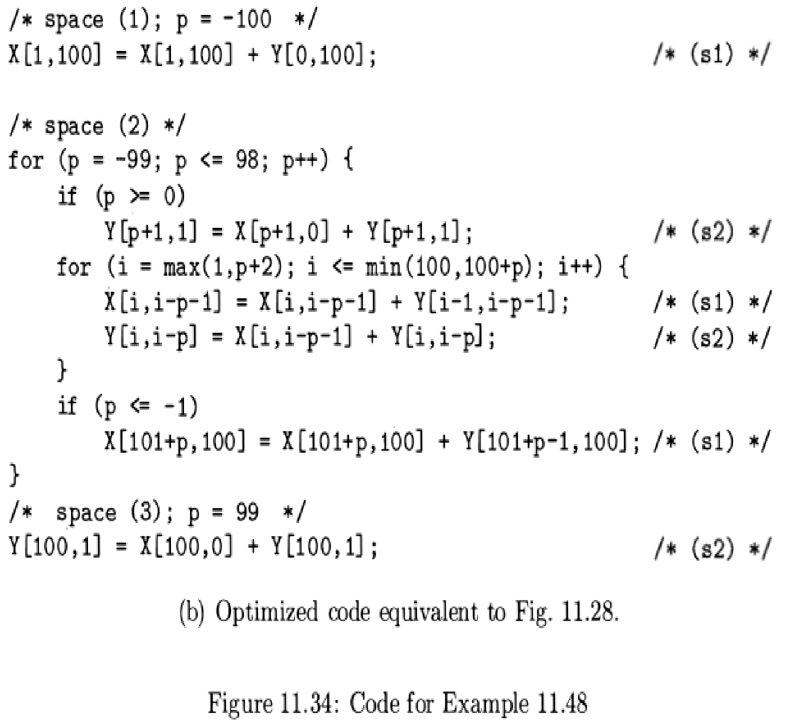


```
/* space (1); p = -100  */
X[1,100] = X[1,100] + Y[0,100];                         /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1];                 /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1];         /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p];               /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}
/*  space (3); p = 99  */
Y[100,1] = X[100,0] + Y[100,1];                         /* (s2) */
```

(b) Optimized code equivalent to Fig. 11.28.

Figure 11.34: Code for Example 11.48

# Code generation – MPMD (Multiple Program Multiple Data stream)



```
/* space (1); p = -100  */
X[1,100] = X[1,100] + Y[0,100];                    /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1];            /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1];    /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p];          /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}
/*  space (3); p = 99  */
Y[100,1] = X[100,0] + Y[100,1];                    /* (s2) */
```

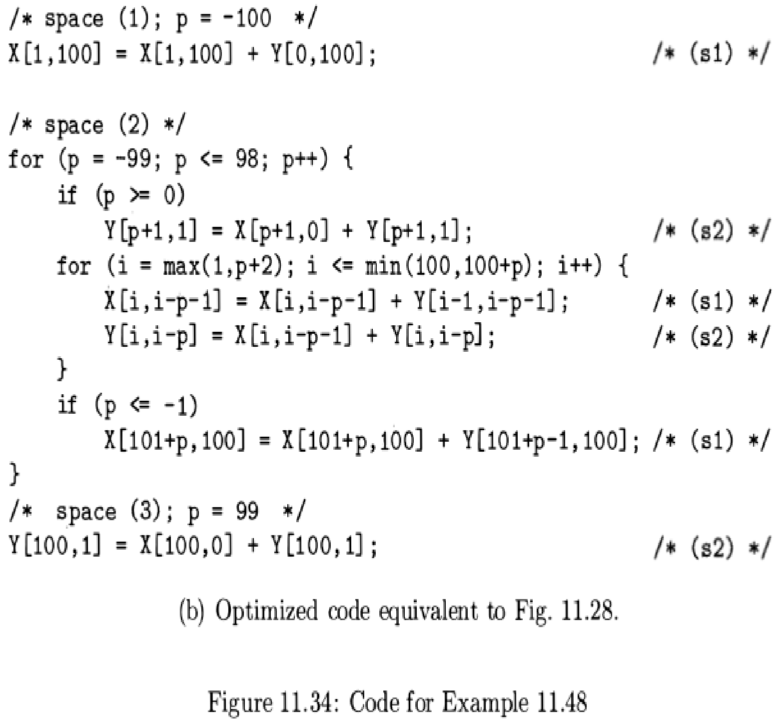
(b) Optimized code equivalent to Fig. 11.28.

Figure 11.34: Code for Example 11.48

```
// processor[0]
/* space ( 1 ) ; p = - 100 */
X[1, 100] = X[1, 100 ] + Y[0, 100];                          /* (s1) */

/* space ( 2 ) */
for (p = -99; p <= -50; p++) {
  for (i = max(1 , p+2); i <= min(100 , 100+p) ; i++) {
    X[i, i-p-1] = X[i, i-p-1] + Y[i-1, i-p-1] ;              /* (s1) */
    Y[i, i-p] = X[i, i-p-1] + Y[i, i-p] ;                    /* (s2) */
  }
  if (p <= -1)
    X[101+p, 100] = X[101+p, 100] + Y[101+p-1, 100]; /* (s1) */
}
```

```
// processor[7]
/* space ( 2 ) */
for (p = 51; p <= 99; p++) {
  if (p >= 0)
    Y[p+1, 1] = X[p+1 , 0] + Y[p+1 , 1];                     /* (s2) */
  for (i = max(1 , p+2); i <= min(100 , 100+p) ; i++) {
    X[i, i-p-1] = X[i, i-p-1] + Y[i-1, i-p-1] ;              /* (s1) */
    Y[i, i-p] = X[i, i-p-1] + Y[i, i-p] ;                    /* (s2) */
  }
}
/* space (3) ; p = 99 */
Y[100, 1] = X[100, 0] + Y[100, 1];
```

# Code generation – SPMD



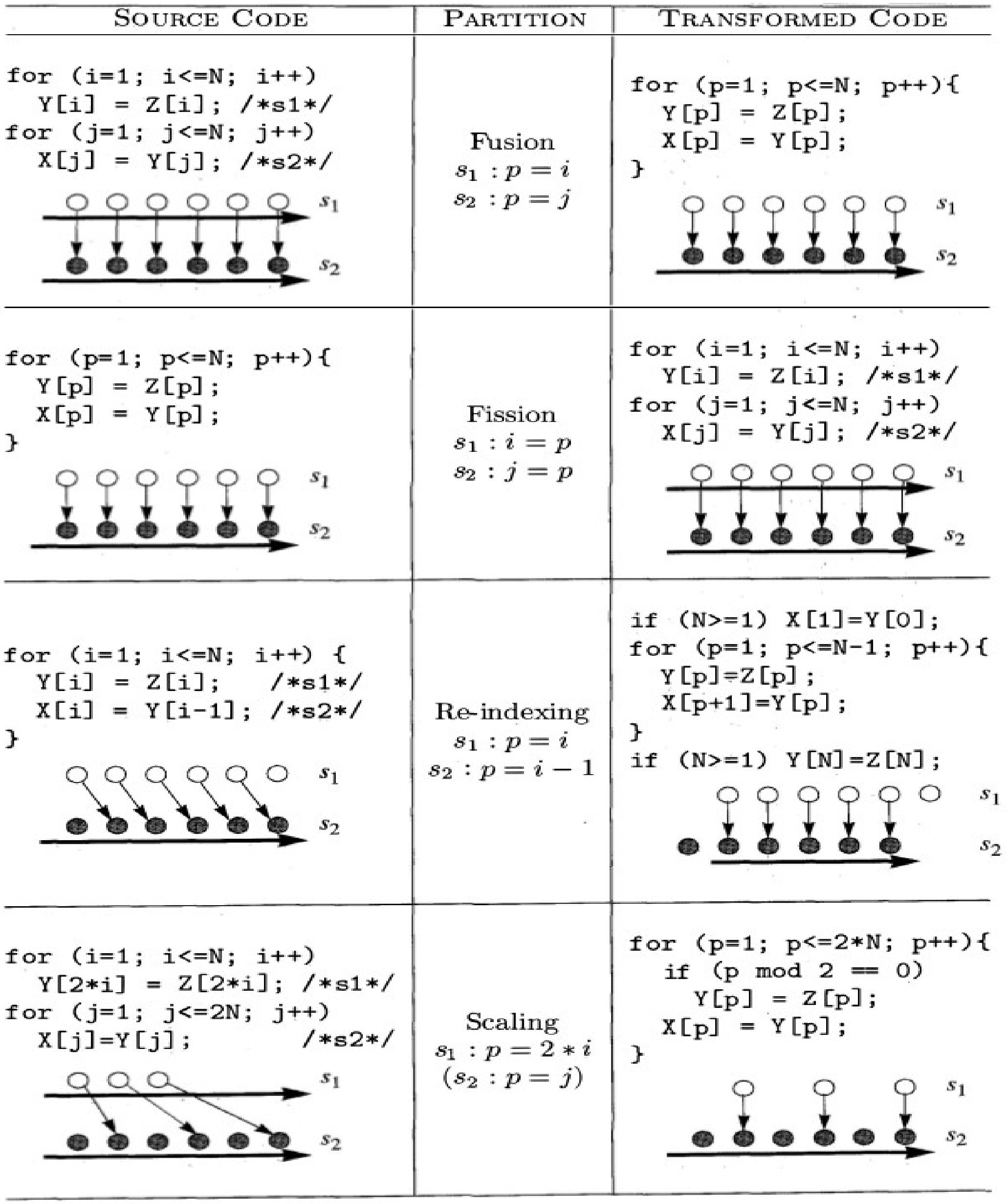
```
/* space (1); p = -100  */
X[1,100] = X[1,100] + Y[0,100];                       /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1];               /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1];       /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p];             /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}
/*  space (3); p = 99  */
Y[100,1] = X[100,0] + Y[100,1];                       /* (s2) */
```

(b) Optimized code equivalent to Fig. 11.28.

Figure 11.34: Code for Example 11.48

```
void AssignProcess() {
  processor[0].assignProcess(-100, -99, -98, ...);
  processor[1].assignProcess(-59, -58, -57, ...);
  …
}


if (processor[pId].hasProcess(-100)) // pId: processor Id
  /* space ( 1 ) ; p = - 100 */
  X[1, 100] = X[1, 100 ] + Y[0, 100];                          /* (s1) */

/* space ( 2 ) */
for (it=processor[pId].begin(); it != processor[pId].end(); it++) {
  p = *it;
  if (p >= 0)
    Y[p+1, 1] = X[p+1 , 0] + Y[p+1 , 1];                        /* (s2) */
  for (i = max(1 , p+2); i <= min(100 , 100+p) ; i++) {
    X[i, i-p-1] = X[i, i-p-1] + Y[i-1, i-p-1] ;                 /* (s1) */
    Y[i, i-p] = X[i, i-p-1] + Y[i, i-p] ;                       /* (s2) */
  }
  if (p <= -1)
    X[101+p, 100] = X[101+p, 100] + Y[101+p-1, 100]; /* (s1) */
}
If (processor[pId].hasProcess(99))
  /* space (3) ; p = 99 */
  Y[100, 1] = X[100, 0] + Y[100, 1] ;                          /* (s2) */
```

# Primitive affine transforms



SOURCE CODE | PARTITION | TRANSFORMED CODE

```
for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/
```

Fusion
$s_1 : p = i$
$s_2 : p = j$

```
for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}
```

```
for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}
```

Fission
$s_1 : i = p$
$s_2 : j = p$

```
for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/
```

```
for (i=1; i<=N; i++) {
  Y[i] = Z[i];    /*s1*/
  X[i] = Y[i-1]; /*s2*/
}
```

Re-indexing
$s_1 : p = i$
$s_2 : p = i - 1$

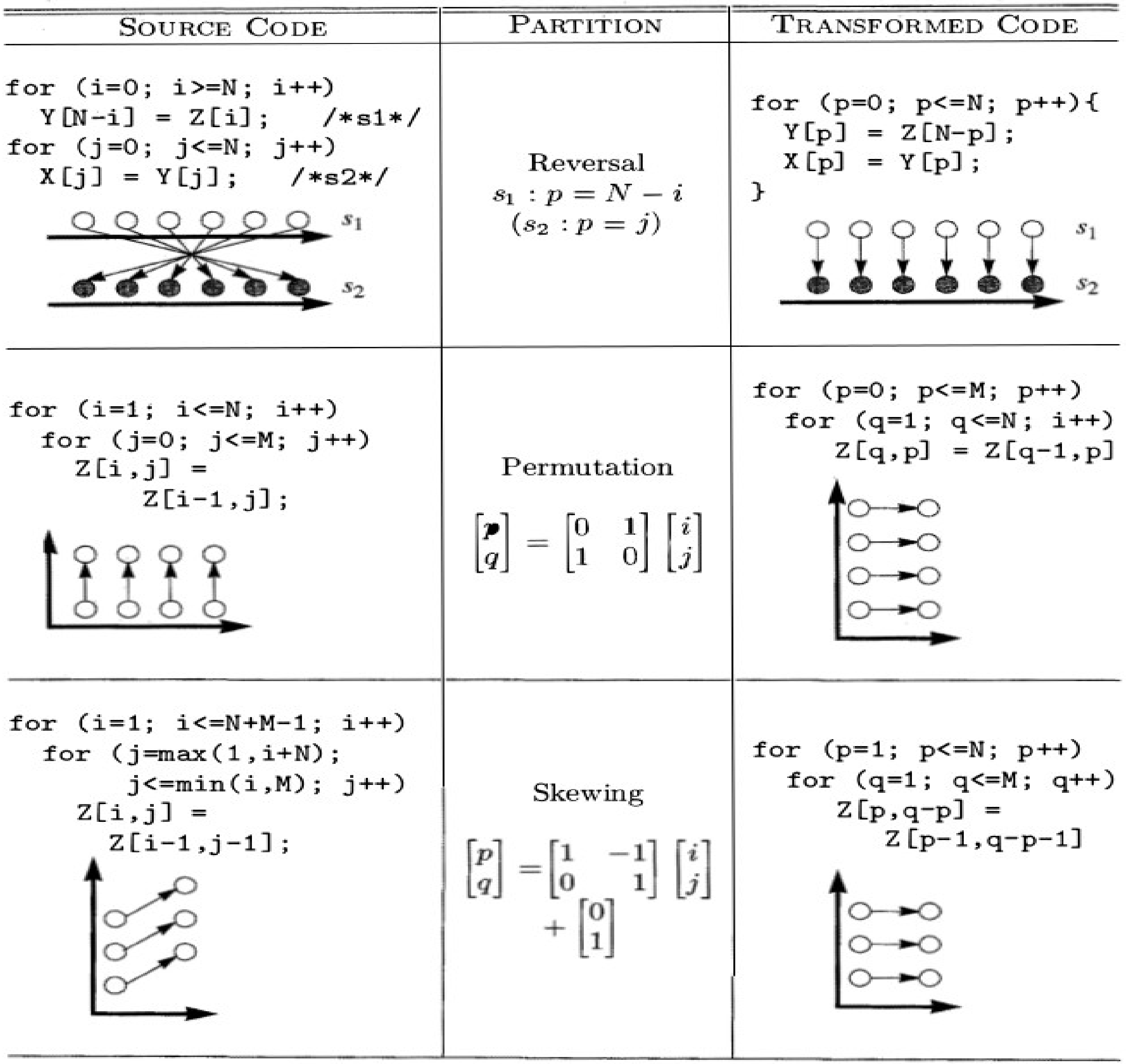
```
if (N>=1) X[1]=Y[0];
for (p=1; p<=N-1; p++){
  Y[p]=Z[p];
  X[p+1]=Y[p];
}
if (N>=1) Y[N]=Z[N];
```

```
for (i=1; i<=N; i++)
  Y[2*i] = Z[2*i]; /*s1*/
for (j=1; j<=2N; j++)
  X[j]=Y[j];        /*s2*/
```

Scaling
$s_1 : p = 2 * i$
$(s_2 : p = j)$

```
for (p=1; p<=2*N; p++){
  if (p mod 2 == 0)
    Y[p] = Z[p];
  X[p] = Y[p];
}
```

# Primitive affine transforms



SOURCE CODE | PARTITION | TRANSFORMED CODE

**SOURCE CODE**

```
for (i=0; i>=N; i++)
  Y[N-i] = Z[i];    /*s1*/
for (j=0; j<=N; j++)
  X[j] = Y[j];    /*s2*/
```

**PARTITION**

Reversal
$s_1 : p = N - i$
$(s_2 : p = j)$

**TRANSFORMED CODE**

```
for (p=0; p<=N; p++){
  Y[p] = Z[N-p];
  X[p] = Y[p];
}
```

**SOURCE CODE**

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

**PARTITION**

Permutation

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

**TRANSFORMED CODE**

```
for (p=0; p<=M; p++)
  for (q=1; q<=N; i++)
     Z[q,p] = Z[q-1,p]
```

**SOURCE CODE**

```
for (i=1; i<=N+M-1; i++)
  for (j=max(1,i+N);
       j<=min(i,M); j++)
    Z[i,j] =
      Z[i-1,j-1];
```

**PARTITION**

Skewing

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

**TRANSFORMED CODE**

```
for (p=1; p<=N; p++)
  for (q=1; q<=M; q++)
     Z[p,q-p] =
        Z[p-1,q-p-1]
```

Figure 11.36: Primitive affine transforms (II)

This example is error

# Synchronization Between Parallel Loops

**Example 11.50 :** Consider the following loop:

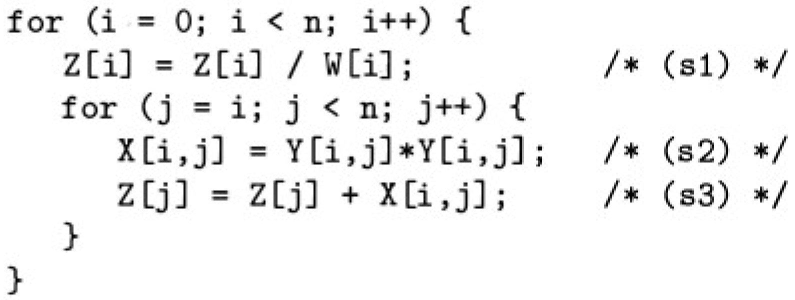
```
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];    /* (s1) */
    W[A[i]] = X[i];        /* (s2) */
}
```

Figure 11.40: Program-dependence graph for the program of Example 11.50

```
X[p] = Y[p] + Z[p];    /* (s1) */
/* synchronization barrier */
if (p == 0)
    for (i=1; i<=n; i++)
        W[A[i]] = X[i];  /* (s2) */
```

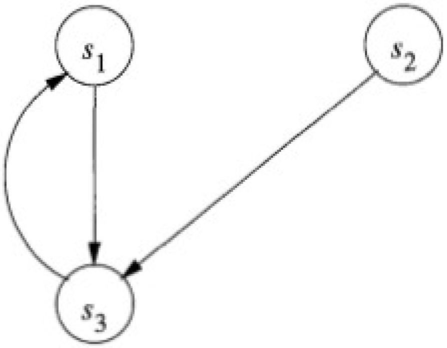
Figure 11.39: SPMD code for the loop in Example 11.50, with $p$ being a variable holding the processor ID

# Synchronization Between Parallel Loops



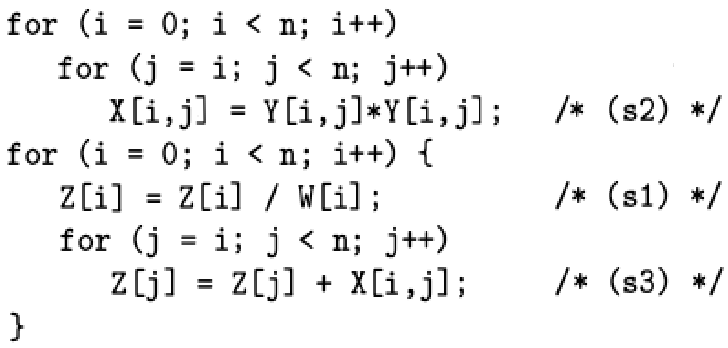
```
for (i = 0; i < n; i++) {
   Z[i] = Z[i] / W[i];            /* (s1) */
   for (j = i; j < n; j++) {
      X[i,j] = Y[i,j]*Y[i,j];     /* (s2) */
      Z[j] = Z[j] + X[i,j];       /* (s3) */
   }
}
```

(a) A program.



(b) Its dependence graph.

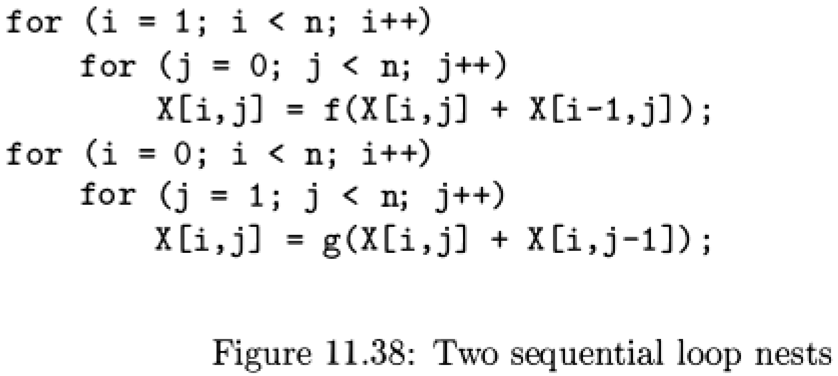
Figure 11.41: Program and dependence graph for Example 11.52.



```
for (i = 0; i < n; i++)
   for (j = i; j < n; j++)
      X[i,j] = Y[i,j]*Y[i,j];   /* (s2) */
for (i = 0; i < n; i++) {
   Z[i] = Z[i] / W[i];          /* (s1) */
   for (j = i; j < n; j++)
      Z[j] = Z[j] + X[i,j];     /* (s3) */
}
```

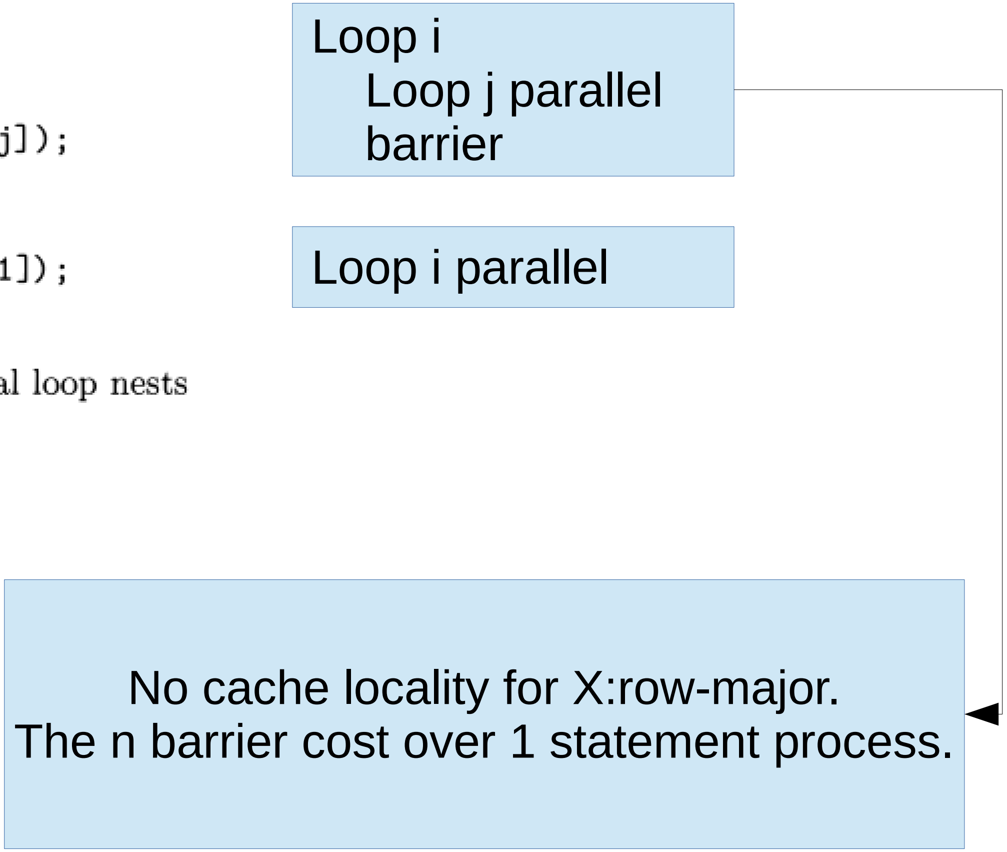
Figure 11.42: Grouping strongly connected components of a loop nest

# Synchronization Between Parallel Loops



```
for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
for (i = 0; i < n; i++)
    for (j = 1; j < n; j++)
        X[i,j] = g(X[i,j] + X[i,j-1]);
```

Figure 11.38: Two sequential loop nests

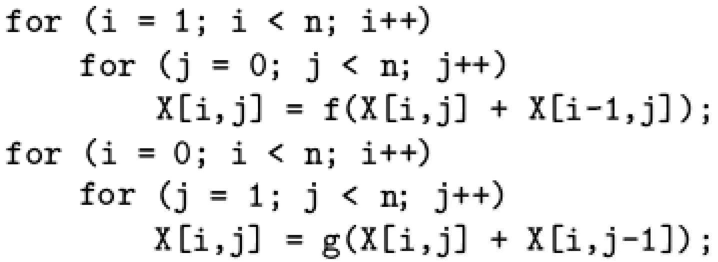


Loop i
    Loop j parallel
    barrier

Loop i parallel

No cache locality for X:row-major.
The n barrier cost over 1 statement process.

# Synchronization Between Parallel Loops



```
for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
for (i = 0; i < n; i++)
    for (j = 1; j < n; j++)
        X[i,j] = g(X[i,j] + X[i,j-1]);
```

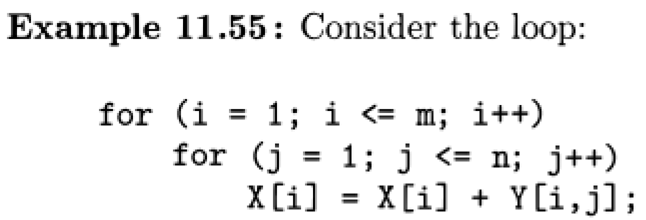
Figure 11.38: Two sequential loop nests

```
for (i = 0; i < n; i++)
   for ( j = 0; j < n; j++)
      $X^T[j,i]=f(X^T[j,i]+X^T[j,i-1]);$
```

```
for (j = 0; j < n; j++)
   for ( i = 0; i < n; i++)
      $X^T[j,i]=f(X^T[j,i]+X^T[j,i-1]);$
```

1 barrier as well as cache locality.
Cost with transpose X.

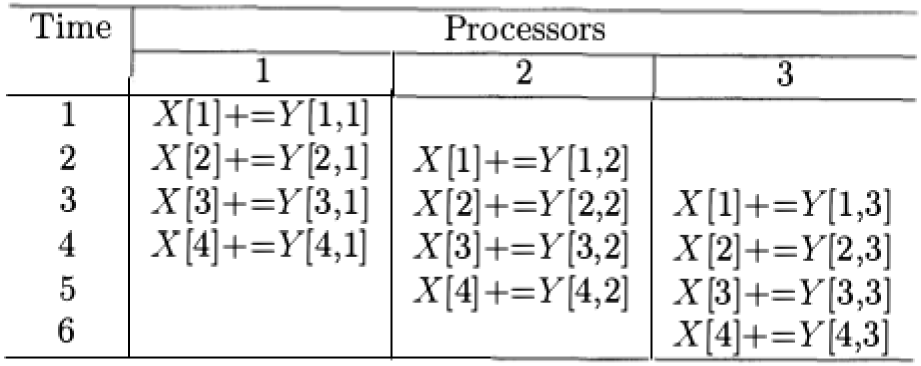
# Pipelining



**Example 11.55:** Consider the loop:

```
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];
```

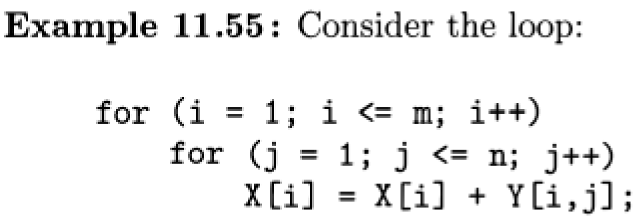
P1: X[1]+=Y[1,1], X[1]+=Y[1,2], X[1]+=Y[1,3]
P2: X[2]+=Y[2,1], X[2]+=Y[2,2], X[2]+=Y[2,3]
...



| Time | Processors | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| 1 | $X[1]{+}{=}Y[1,1]$ | | |
| 2 | $X[2]{+}{=}Y[2,1]$ | $X[1]{+}{=}Y[1,2]$ | |
| 3 | $X[3]{+}{=}Y[3,1]$ | $X[2]{+}{=}Y[2,2]$ | $X[1]{+}{=}Y[1,3]$ |
| 4 | $X[4]{+}{=}Y[4,1]$ | $X[3]{+}{=}Y[3,2]$ | $X[2]{+}{=}Y[2,3]$ |
| 5 | | $X[4]{+}{=}Y[4,2]$ | $X[3]{+}{=}Y[3,3]$ |
| 6 | | | $X[4]{+}{=}Y[4,3]$ |

Figure 11.49: Pipelined execution of Example 11.55 with $m = 4$ and $n = 3$.

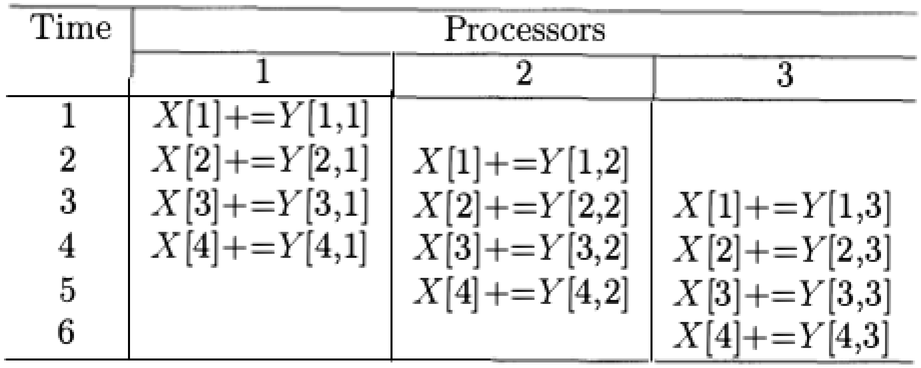
# Pipelining



**Example 11.55:** Consider the loop:

```
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];
```

P1: X[1]+=Y[1,1], X[1]+=Y[1,2], X[1]+=Y[1,3]
P2: X[2]+=Y[2,1], X[2]+=Y[2,2], X[2]+=Y[2,3]
...



| Time | Processors | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| 1 | $X[1] {+}{=} Y[1,1]$ | | |
| 2 | $X[2] {+}{=} Y[2,1]$ | $X[1] {+}{=} Y[1,2]$ | |
| 3 | $X[3] {+}{=} Y[3,1]$ | $X[2] {+}{=} Y[2,2]$ | $X[1] {+}{=} Y[1,3]$ |
| 4 | $X[4] {+}{=} Y[4,1]$ | $X[3] {+}{=} Y[3,2]$ | $X[2] {+}{=} Y[2,3]$ |
| 5 | | $X[4] {+}{=} Y[4,2]$ | $X[3] {+}{=} Y[3,3]$ |
| 6 | | | $X[4] {+}{=} Y[4,3]$ |

Figure 11.49: Pipelined execution of Example 11.55 with $m = 4$ and $n = 3$.

When Y is stored in column-major
→ locality in the same processor.

Beside treat processors as the core of CPU,
it can be function unit in CPU.

# 3 Pipeline – superscaler

```
for ( i = 1 ; i <= m ; i++)
  for ( j = 1 ; j <= 3; j++) {
    X[i] = X[i] + Y [i , j ] ;
    X[i] = X[i] * Y [i , j ] ;
    X[i] = X[i] - Y [i , j ] ;
  }
```

```
for ( i = 1 ; i <= 3 ; i++)
  X[i] = X[i] + Y [i , 1 ] ;
for ( i = 4 ; i <= 6 ; i++)
  X[i] = X[i] + Y [i , 1 ] ;
for ( i = 1 ; i <= 3 ; i++)
  X[i] = X[i] * Y [i , 2 ] ;
for ( i = 7 ; i <= m ; i++) {
  for ( j = 3 ; j <= n; j++) {
    X[i] = X[i] + Y [i , j-2 ] ;
    X[i-1] = X[i-1] * Y [i-1 , j-1 ] ;
    X[i-2] = X[i-2] - Y [i-2 , j ] ;
  }
...
```

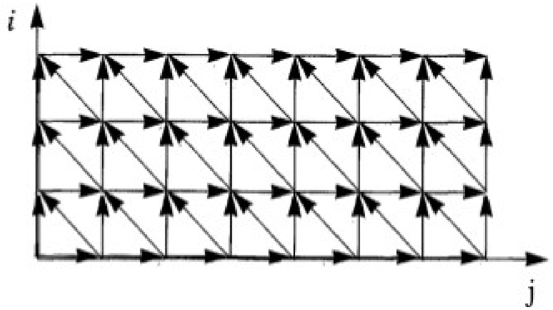
| Time | pipeline | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| 1 | X[1]+=Y[1,1] | | |
| 1 | X[2]+=Y[2,1] | | |
| 1 | X[3]+=Y[3,1] | | |
| 2 | X[4]+=Y[4,1] | X[1]*=Y[1,2] | |
| 2 | X[5]+=Y[5,1] | X[2]*=Y[2,2] | |
| 2 | X[6]+=Y[6,1] | X[3]*=Y[3,2] | |
| 3 | X[7]+=Y[7,1] | X[4]*=Y[4,2] | X[1]-=Y[1,3] |
| 3 | X[8]+=Y[8,1] | X[5]*=Y[5,2] | X[2]-=Y[2,3] |
| 3 | | X[6]*=Y[6,2] | X[3]-=Y[3,3] |
| 4 | | X[7]*=Y[7,2] | X[4]-=Y[4,3] |
| 4 | | X[8]*=Y[8,2] | X[5]-=Y[5,3] |
| 4 | | | X[6]-=Y[6,3] |
| 5 | | | X[7]-=Y[7,3] |
| 5 | | | X[8]-=Y[8,3] |

SIMD instruction

# Pipelining

```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

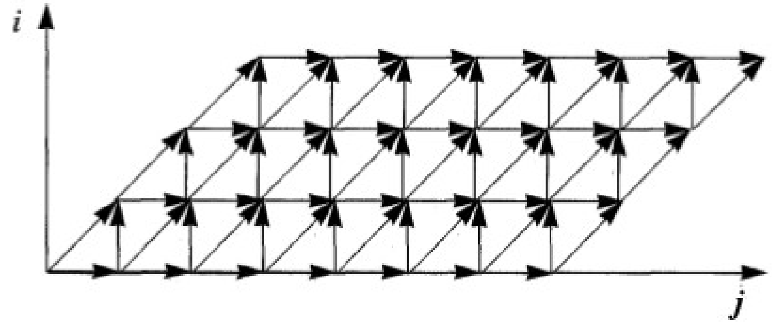


(b) Data dependences in the code.

Figure 11.50: An example of successive over-relaxation (SOR)

```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

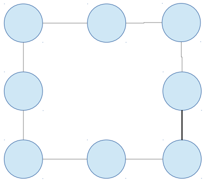
(a) The code in Fig. 11.50 transformed by $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.



(b) Data dependences of the code in (a).

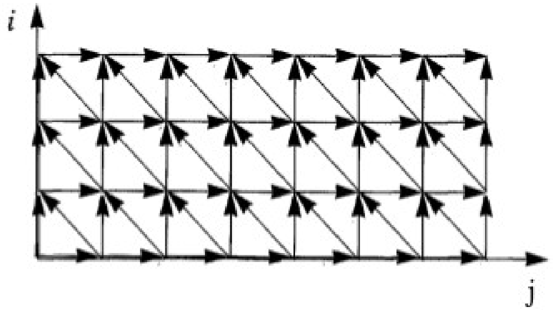
# Pipelining

Linear, Ring above (mesh, cube) connected processors.



```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

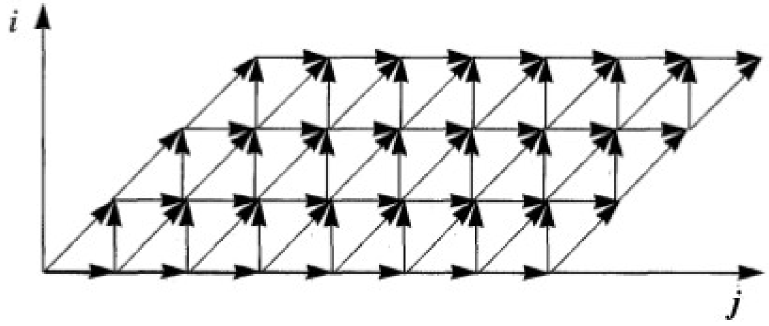


(b) Data dependences in the code.

Figure 11.50: An example of successive over-relaxation (SOR)

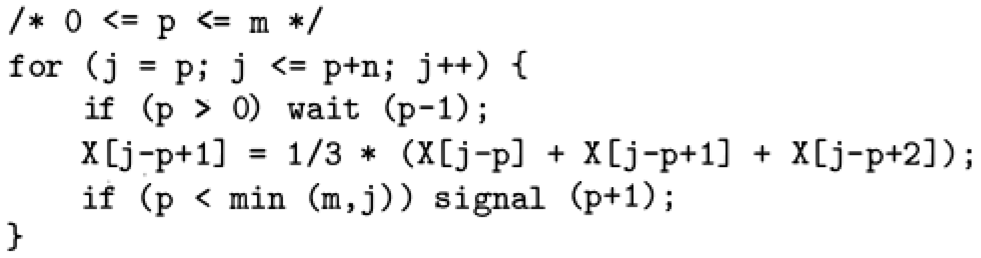
```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

(a) The code in Fig. 11.50 transformed by $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.



(b) Data dependences of the code in (a).

# Pipelining - synchronize



```
/* 0 <= p <= m */
for (j = p; j <= p+n; j++) {
    if (p > 0) wait (p-1);
    X[j-p+1] = 1/3 * (X[j-p] + X[j-p+1] + X[j-p+2]);
    if (p < min (m,j)) signal (p+1);
}
```

(a) Processors assigned to rows.

# Pipelining – linear connection – verify

```
if (p > 0) wait (p-1 ) ;
X[ j-p+1] = 1/3 * (X[ j-p]+X[ j-p+1]+X[ j-p+2]);
if (p < min(m , j ) ) signal(p+1) ;
```

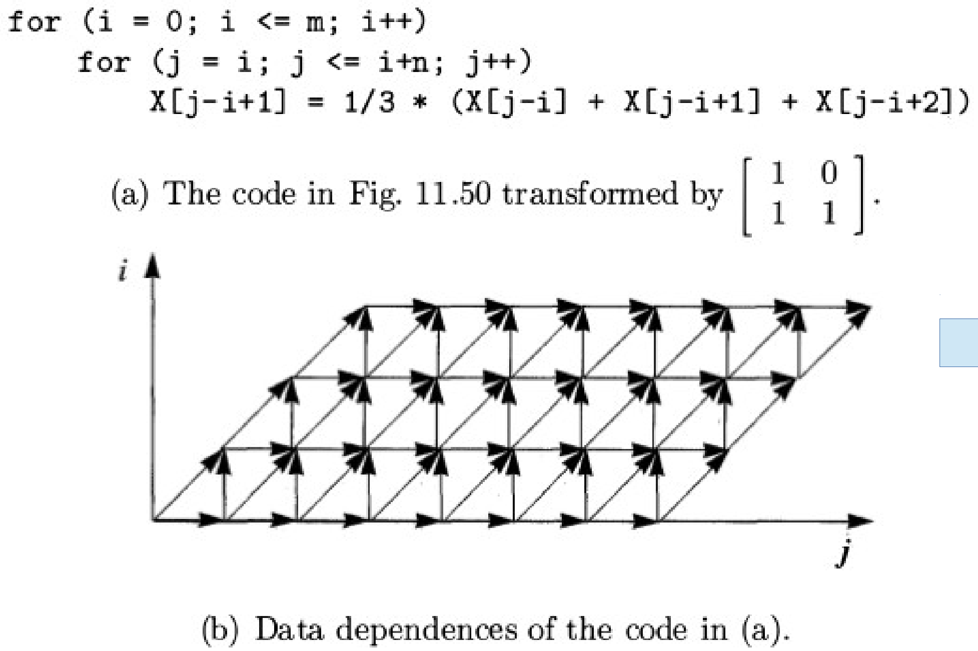
```
p=0; i=0; j=1;
if (p > 0) wait (p-1 ) ;
X[1-0+1] = 1/3 * (X[1-0]+X[1-0+1]+X[1-0+2]);
if (p < min(m , j ) ) signal(p+1) ;
```

```
p=1; i=1; j=1;
if (p > 0) wait (p-1 ) ;
X[1-1+1] = 1/3 * (X[1-1]+X[1-1+1]+X[1-1+2]);
if (p < min(m , j ) ) signal(p+1) ;
```

```
for ( i = 0 ; i <= m ; i++)
  for ( j = i ; j <= i+n ; j ++)
    X[ j-i+1] = 1/3 * (X[ j-i] + X[ j-i+1] + X[ j-i+2]);
```

```
(i, j):
(0,0)
  X[1] = 1/3 * (X[0]+X[1]+X[2]);
  (0,1)
    X[2] = 1/3 * (X[1]+X[2]+X[3]);
    (0,2)
      X[3] = 1/3 * (X[2]+X[3]+X[4]);

  (1,1)
    X[1] = 1/3 * (X[0]+X[1]+X[2]);
```

# Pipelining

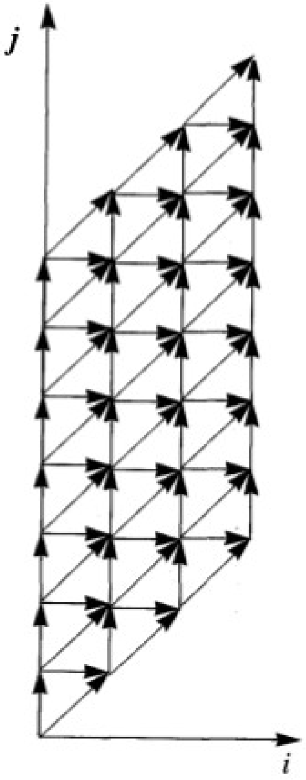


```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

(a) The code in Fig. 11.50 transformed by $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

(b) Data dependences of the code in (a).

```
for (j = 0; j <= m+n; i++)
    for (i = max(0,j-n); i <= min(m,j); j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2];
```



(d) Data dependences of the code in (b).

# Pipelining formulate – variable space, data dependence & time steps mapping

If there exists a data-dependent pair of instances, $i_1$ of $s_1$ and $i_2$ of $s_2$, and $i_1$ is executed before $i_2$ in the original program.

- For all $\mathbf{i}_1$ in $Z^{d_1}$ and $\mathbf{i}_2$ in $Z^{d_2}$ such that

  a) $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$,
  b) $\mathbf{B}_1\mathbf{i}_1 + \mathbf{b}_1 \geq 0$,
  c) $\mathbf{B}_2\mathbf{i}_2 + \mathbf{b}_2 \geq 0$, and
  d) $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$,

  it is the case that $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 \leq \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$.

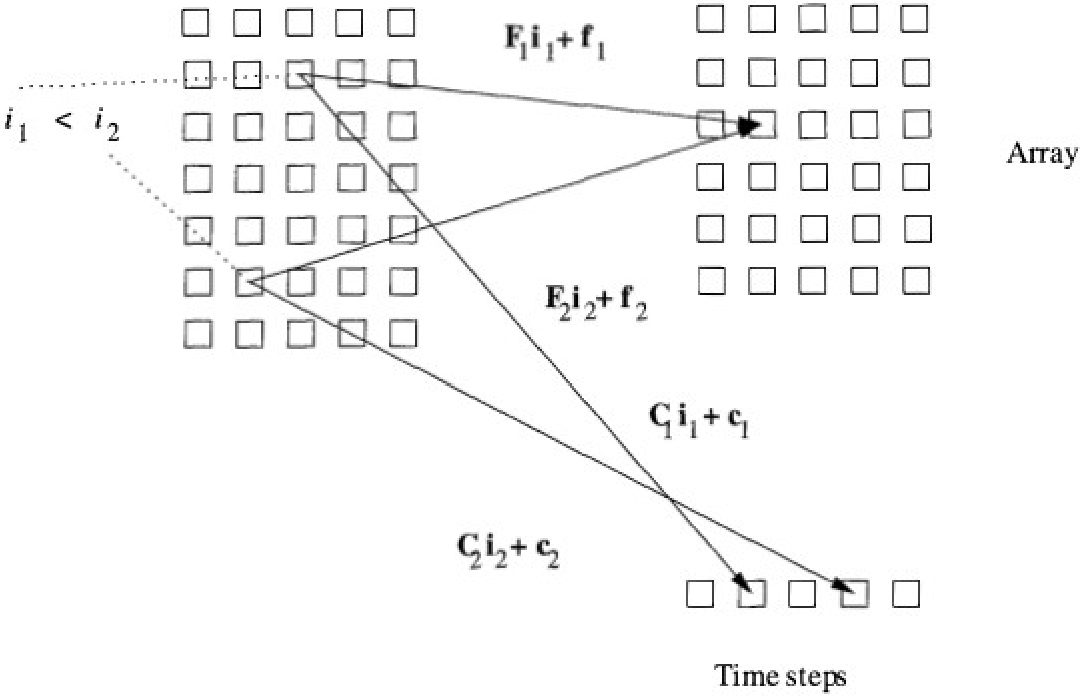
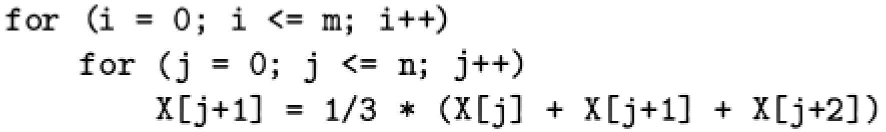


Figure 11.54: Time-Partition Constraints

# Pipelining – time steps mapping



```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

$(i\quad j) \prec_{s_1 s_2} (i'\quad j') \rightarrow (i\quad j) \prec (i'\quad j')$

consider all processors have the same time steps mapping function

$[C_1\quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] \le [C_1\quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$

if there is a dependence from $(i, j)$ to $(i', j')$. By definition, $(i\quad j) \prec (i'\quad j')$ is, either $i < i'$ or $(i = i' \wedge j < j')$

1 . True dependence from write access X[ j + 1] to read access X[ j + 2]. eg. X[1+1](i=0,j=1), X[0+2](i'=1,j'=0).

$j+1 = j'+2 \rightarrow j = j'+1$

$C_1(i'-i) - C_2 \ge 0$

$C_1 - C_2 \ge 0 \,(since\; j > j' \rightarrow i < i')$

2. Antidependence from read access X[ j + 2] to write access X[ j + 1] . eg. X[0+2](i=1,j=0), X[1+1](i'=1,j'=1).

$j+2 = j'+1 \rightarrow j = j'-1$

$C_1(i'-i) + C_2 \ge 0$

$C_2 \ge 0 \,(when\; i = i')$

$C_1 \ge 0 \,(since\; C_2 \ge 0, when\; i < i')$

3. Output dependence from write access X[ j + 1] back to itself. eg. X[0+1](i=1,j=0), X[1+0](i'=2,j'=0).

$j = j'$

$C_1(i'-i) \ge 0$

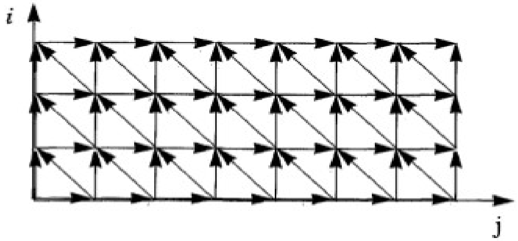
$C_1 \ge 0 \,(since\; i < i')$

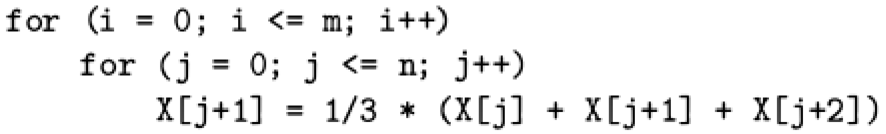
The rest of the dependences do not yield any new constraints.

Conclusion:

$C_1 \ge 0, C_2 \ge 0, C_1 - C_2 \ge 0 \rightarrow [1\quad 0], [1\quad 1]$

# Pipelining – Code generation





```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

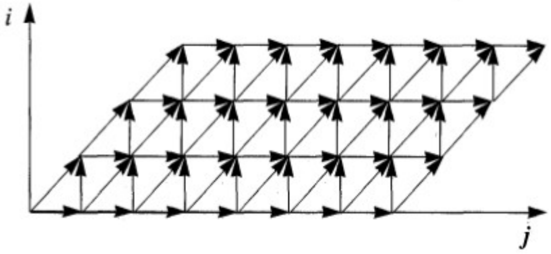
$$\begin{bmatrix}i'\\j'\end{bmatrix}=\begin{bmatrix}C_{11}&C_{12}\\C_{21}&C_{22}\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}=\begin{bmatrix}1&0\\1&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}$$

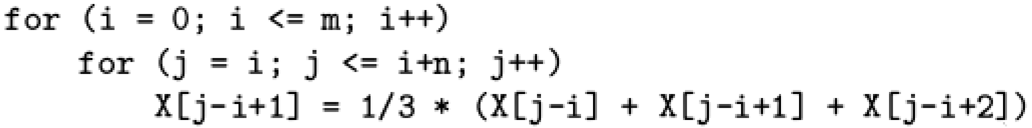
$$\rightarrow i'=i, j'=\begin{bmatrix}1&1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}=i+j$$

$$\rightarrow j=j'-i$$

```
for (i' = 0; i' <= m; i'++)
    for (j' = i'; j' <= i'+n; j'++)
        X[ j'-i'+1] = 1/3 * (X[ j'-i'] + X[ j'-i'+1] + X[ j'-i'+2];
```

$i' \leftarrow i, j' \leftarrow j$





```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

(a) The code in Fig. 11.50 transformed by $\begin{bmatrix}1&0\\1&1\end{bmatrix}$.

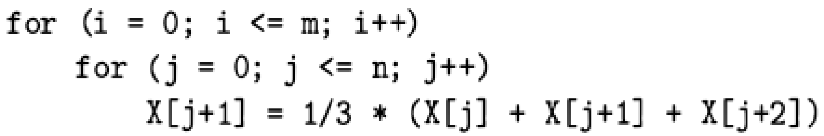
# Pipelining – Code generation

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

$\rightarrow i' = [1 \quad 1] \begin{bmatrix} i \\ j \end{bmatrix} = i + j, j' = i$

$\rightarrow j = i' - i = i' - j'$

$\rightarrow 0 \le j \le n, 0 \le i' - j' \le n$



```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

```
for (i' = 0; i' <= m+n; i'++)
  for (j' = 0; j' <= m; j'++) {
    if ((i'-j' >= 0)&&(i'-j' <=n))
      X[i'-j'+1] = 1/3 * (X[i'-j'] + X[i'-j'+1] + X[i'-j'+2];
  }
```

$0 \le j' \le m$

$0 \le i' - j' \le n \rightarrow -i' \le -j' \le n - i' \rightarrow i' - n \le j' \le i'$

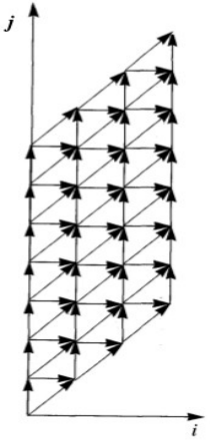
```
for (i' = 0; i' <= m+n; i'++)
    for (j' = max(0,i'-n); j <= min(m,i'); j'++)
        X[i'-j'+1] = 1/3 * (X[i'-j'] + X[i'-j'+1] + X[i'-j'+2];
```

$i' \leftarrow j, j' \leftarrow i$

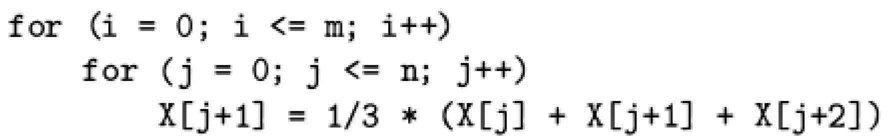
```
for (j = 0; j <= m+n; j++)
    for (i = max(0, j-n); i <= min(m, j); i++)
        X[ j-i+1] = 1/3 * (X[ j-i] + X[ j-i+1] + X[ j-i+2];
```

m=3,n=6

j=0 → i=0
 =1 → i=0,1
 =2 → i=0,1,2
 =3 → i=0,1,2,3
 =4 → i=0,1,2,3
 =5 → i=0,1,2,3
 =6 → i=0,1,2,3
 =7 → i=1,2,3
 =8 → i=2,3
 =9 → i=3



# Pipelining – time steps mapping – function unit



```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

$(i \quad j) \prec_{s_1 s_2} (i' \quad j') \rightarrow (i \quad j) \prec (i' \quad j')$

consider all processors have the same time steps mapping function

true and output dependence $[C_1 \quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] < [C_1 \quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$ antidependence $[C_1 \quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] \leq [C_1 \quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$

if there is a dependence from $(i, j)$ to $(i', j')$. By definition, $(i \quad j) \prec (i' \quad j')$ is, either $i < i'$ or $(i = i' \wedge j < j')$

1 . True dependence from write access X[ j + 1] to read access X[ j + 2]. eg. X[1+1](i=0,j=1), X[0+2](i'=1,j'=0).

$$j+1 = j'+2 \rightarrow j = j'+1$$
$$C_1(i'-i) - C_2 > 0$$
$$C_1 - C_2 > 0 \,(since\ j > j' \rightarrow i < i')$$

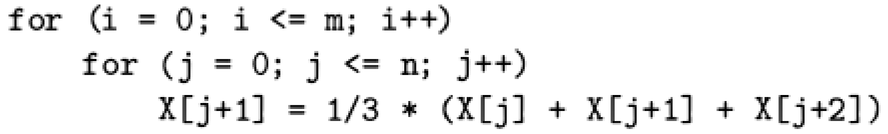
2. Antidependence from read access X[ j + 2] to write access X[ j + 1] . eg. X[0+2](i=1,j=0), X[1+1](i'=1,j'=1).

$$j+2 = j'+1 \rightarrow j = j'-1$$
$$C_1(i'-i) + C_2 \geq 0$$
$$C_2 \geq 0 \,(when\, i = i')$$
$$C_1 \geq 0 \,(since\, C_2 \geq 0, when\, i < i')$$

3. Output dependence from write access X[ j + 1] back to itself. eg. X[0+1](i=1,j=0), X[1+0](i'=2,j'=0).

$$j = j'$$
$$C_1(i'-i) > 0$$
$$C_1 > 0 \,(since\, i < i')$$

# Pipelining – time steps mapping – function unit



```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

$(i\quad j) \prec_{s_1 s_2} (i'\quad j') \rightarrow (i\quad j) \prec (i'\quad j')$

consider all processors have the same time steps mapping function

true and output dependence $[C_1\quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] < [C_1\quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$ antidependence $[C_1\quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] \leq [C_1\quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$

if there is a dependence from $(i, j)$ to $(i', j')$. By definition, $(i\quad j) \prec (i'\quad j')$ is, either $i < i'$ or $(i = i' \wedge j < j')$

4 . True dependence from write access X[ j + 1] to read access X[ j + 1]. eg. X[1+1](i=0,j=1), X[0+2](i'=1,j'=1).

$j + 1 = j' + 1 \rightarrow j = j'$
$C_1(i' - i) > 0$
$C_1 > 0 (since\ j = j' \rightarrow i < i')$

5. Antidependence from read access X[ j + 1] to write access X[ j + 1] . eg. X[0+2](i=1,j=1), X[1+1](i'=1,j'=1).

$j + 1 = j' + 1 \rightarrow j = j'$
$C_1(i' - i) \geq 0$
$C_1 \geq 0 (when\ i < i')$

6 . True dependence from write access X[ j + 1] to read access X[ j ]. eg. X[1+1](i=0,j=1), X[0+2](i'=0,j'=2).

$j + 1 = j' \rightarrow j = j' - 1$
$C_1(i' - i) + C_2 > 0$
$C_2 > 0 (when\ i = i')$

7. Antidependence from read access X[ j ] to write access X[ j + 1] . eg. X[0+2](i=0,j=1), X[1+1](i'=1,j'=0).

$j = j' + 1$
$C_1(i' - i) - C_2 \geq 0$
$C_1 - C_2 \geq 0 (since\ i < i')$

# Pipelining – time steps mapping – function unit

✓ Conclusion:

$C_1 > 0, C_2 > 0, C_1 - C_2 > 0 \rightarrow [2 \quad 1], [3 \quad 2]$

Two solution meaning can pipelining, we take:

$[1 \quad 0]$ (keep i as outmost for function unit number), $[2 \quad 1]$ ( j as time steps mapping function)

✗ The other Conclusion:

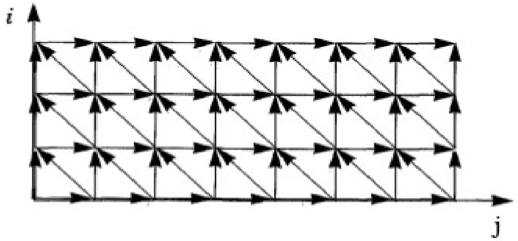
$C_1 > 0, C_2 > 0, C_1 - C_2 > 0 \rightarrow [2 \quad 1], [4 \quad 3]$

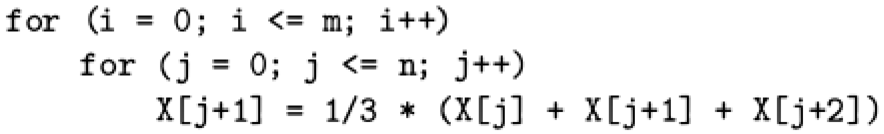
Two solution meaning can pipelining, we take:

$2\mathrm{i} + j = i' = [1 \quad 0]\begin{bmatrix} i' \\ j \end{bmatrix}$ (keep i as outmost for function unit number)

$4\mathrm{i} + 3\mathrm{j} = 2\mathrm{i}' + j = [2 \quad 1]\begin{bmatrix} i' \\ j \end{bmatrix}$

# Pipelining – Code generation – function unit





```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

$\rightarrow i' = i, j' = 2\mathrm{i} + j$

$\rightarrow j = j' - 2\mathrm{i}$

```
for (i' = 0; i' <= m; i'++)
   for (j' = 2*i'; j' <= 2*i'+n; j'++)
      X[ j'-2*i'+1] = 1/3 * (X[ j'-2*i'] + X[ j'-2*i'+1] + X[ j'-2*i'+2];
```

$i' \leftarrow i, j' \leftarrow j$

```
for (i = 0; i <= m; i++)
   for (j = 2*i'; j <= 2*i+n; j++)
      X[ j-2*i+1] = 1/3 * (X[ j-2*i] + X[ j-2*i+1] + X[ j-2*i+2];
```

# Pipelining – function unit – verify

```
for (i = 0; i <= m; i++)
    for (j = 2*i'; j <= 2*i+n; j++)
        X[ j-2*i+1] = 1/3 * (X[ j-2*i] + X[ j-2*i+1] + X[ j-2*i+2];
```

```
(i, j):
(0,0)
  X[1] = 1/3 * (X[0]+X[1]+X[2]);
  (0,1)
    X[2] = 1/3 * (X[1]+X[2]+X[3]);
    (0,2)
      X[3] = 1/3 * (X[2]+X[3]+X[4]);
      (0,3)
        X[4] = 1/3 * (X[3]+X[4]+X[5]);

    (1,2)
      X[1] = 1/3 * (X[0]+X[1]+X[2]);
      (1,3)
        X[2] = 1/3 * (X[1]+X[2]+X[3]);
        (1,4)
          X[3] = 1/3 * (X[2]+X[3]+X[4]);

        (2,4)
          X[1] = 1/3 * (X[0]+X[1]+X[2]);
```

# Pipelining – time steps mapping – function unit – 2

```
for (i = 0; i <= m; i++)
   for ( j = 0; j <= n; j++)
       X[ j] = 1/3 * (X[ j] + X[ j+1] + X[ j+2];
```

$(i \quad j) \prec_{s_1 s_2} (i' \quad j') \rightarrow (i \quad j) \prec (i' \quad j')$

consider all processors have the same time steps mapping function

true and output dependence $[C_1 \quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] < [C_1 \quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$ antidependence $[C_1 \quad C_2]\begin{bmatrix} i \\ j \end{bmatrix} + [c] \le [C_1 \quad C_2]\begin{bmatrix} i' \\ j' \end{bmatrix} + [c]$

if there is a dependence from $(i, j)$ to $(i', j')$. By definition, $(i \quad j) \prec (i' \quad j')$ is, either $i < i'$ or $(i = i' \wedge j < j')$

1 . True dependence from write access X[ j ] to read access X[ j + 2]. eg. X[2](i=0,j=2), X[0+2](i'=1,j'=0).

$j = j' + 2$

$C_1(i' - i) - 2C_2 > 0$

$C_1 - 2C_2 > 0\,(since\ j > j' \rightarrow i < i')$

2. Antidependence from read access X[ j + 2] to write access X[ j ] . eg. X[0+2](i=1,j=0), X[2](i'=1,j'=2).

$j + 2 = j' \rightarrow j = j' - 2$

$C_1(i' - i) + 2C_2 \ge 0$

$2C_2 \ge 0\,(when\, i = i')$

$C_1 > 0\,(since\, C_2 \ge 0, when\, i < i')$

3. Output dependence from write access X[ j ] back to itself. eg. X[0](i=1,j=0), X[0](i'=2,j'=0).

$j = j'$

$C_1(i' - i) > 0$

$C_1 > 0\,(since\, i < i')$

# Pipelining – time steps mapping – function unit – 2

```
for (i = 0; i <= m; i++)
    for ( j = 0; j <= n; j++)
        X[ j] = 1/3 * (X[ j] + X[ j+1] + X[ j+2];
```

$(i \quad j) \prec_{s_1 s_2} (i' \quad j') \rightarrow (i \quad j) \prec (i' \quad j')$

consider all processors have the same time steps mapping function

true and output dependence $[C_1 \quad C_2][\begin{smallmatrix} i \\ j \end{smallmatrix}] + [c] < [C_1 \quad C_2][\begin{smallmatrix} i' \\ j' \end{smallmatrix}] + [c]$ antidependence $[C_1 \quad C_2][\begin{smallmatrix} i \\ j \end{smallmatrix}] + [c] \leq [C_1 \quad C_2][\begin{smallmatrix} i' \\ j' \end{smallmatrix}] + [c]$

if there is a dependence from $(i, j)$ to $(i', j')$. By definition, $(i \quad j) \prec (i' \quad j')$ is, either $i < i'$ or $(i = i' \wedge j < j')$

4 . True dependence from write access X[ j ] to read access X[ j + 1]. eg. X[1+1](i=0,j=2), X[0+2](i'=1,j'=1).

$j = j' + 1$

$C_1(i' - i) - C_2 > 0$

$C_1 - C_2 > 0 \left(since\ j > j' \rightarrow i < i', let\ i = i' - 1\right)$

5. Antidependence from read access X[ j + 1 ] to write access X[ j ] . eg. X[0+2](i=1,j=1), X[1+1](i'=1,j'=2).

$j + 1 = j'$

$C_1(i' - i) + C_2 \geq 0$

$C_2 \geq 0 \left(when\ i = i'\right), C_1 > 0 \left(since\ C_2 \geq 0, when\ i < i'\right)$

6 . True dependence from write access X[ j ] to read access X[ j ]. eg. X[1+1](i=0,j=1), X[0+2](i'=0,j'=2).

$j = j'$

$C_1(i' - i) > 0$

$C_1 > 0 \left(when\ i = i' - 1\right)$

7. Antidependence from read access X[ j ] to write access X[ j ] . eg. X[0+2](i=0,j=1), X[1+1](i'=1,j'=1).

$j = j'$

$C_1(i' - i) \geq 0$

$C_1 \geq 0 \left(when\ i < i'\right)$

# Pipelining – time steps mapping – function unit – 2

Conclusion:

$$C_1 > 0, 2C_2 \geq 0, C_1 - C_2 > 0, C_1 - 2C_2 > 0 \rightarrow [1 \quad 0], [3 \quad 1]$$

# Pipelining – Code generation – function unit – 2

```
for (i = 0; i <= m; i++)
    for ( j = 0; j <= n; j++)
         X[ j] = 1/3 * (X[ j] + X[ j+1] + X[ j+2];
```

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

$$\rightarrow i' = i, j' = 3\mathrm{i} + j$$

$$\rightarrow j = j' - 3\mathrm{i}$$

```
for (i' = 0; i' <= m; i'++)
    for (j' = 3*i'; j' <= 3*i'+n; j'++)
         X[ j'-3*i'] = 1/3 * (X[ j'-3*i'] + X[ j'-3*i'+1] + X[ j'-3*i'+2];
```

$i' \leftarrow i, j' \leftarrow j$

```
for (i = 0; i <= m; i++)
    for (j = 3*i'; j <= 3*i+n; j++)
         X[ j-3*i] = 1/3 * (X[ j-3*i] + X[ j-3*i+1] + X[ j-3*i+2];
```

# Pipelining – function unit – 2 – verify

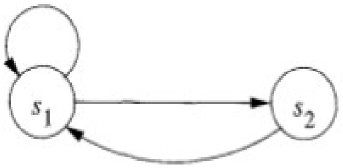
```
for (i = 0; i <= m; i++)
   for ( j = 3*i'; j <= 3*i+n; j++)
       X[ j-3*i] = 1/3 * (X[ j-3*i] + X[ j-3*i+1] + X[ j-3*i+2];
```

```
(i, j):
(0,0)
  X[0] = 1/3 * (X[0]+X[1]+X[2]);
  (0,1)
    X[1] = 1/3 * (X[1]+X[2]+X[3]);
    (0,2)
      X[2] = 1/3 * (X[2]+X[3]+X[4]);
      (0,3)
        X[3] = 1/3 * (X[3]+X[4]+X[5]);

      (1,3)
        X[0] = 1/3 * (X[0]+X[1]+X[2]);
        (1,4)
          X[1] = 1/3 * (X[1]+X[2]+X[3]);
          (1,5)
            X[2] = 1/3 * (X[2]+X[3]+X[4]);
            (1,6)
              X[3] = 1/3 * (X[3]+X[4]+X[5]);

            (2,6)
              X[1] = 1/3 * (X[0]+X[1]+X[2]);
```

# Pipelining – no time steps mapping

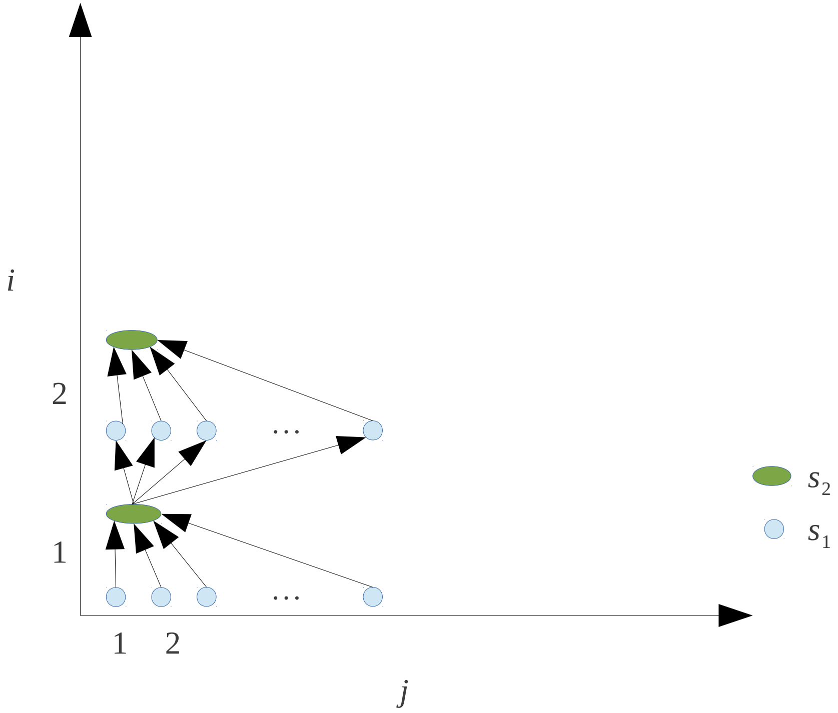
```
for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++)
        X[j] = X[j] + Y[i,j];     /* (s1) */
    Z[i] = X[A[i]];               /* (s2) */
}
```

(a)



(b)

Figure 11.53: A sequential outer loop (a) and its PDG (b)



# Pipelining – no time steps mapping

```
for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++)
        X[j] = X[j] + Y[i,j];     /* (s1) */
    Z[i] = X[A[i]];               /* (s2) */
}
```

1. True dependence from write access X[ j ] to read access X[A[i]]

$[C_{11} \quad C_{12}][\begin{smallmatrix} i \\ j \end{smallmatrix}]+c_1 \le C_{21} i' + c_2$

$\rightarrow C_{11} i + C_{12} j + c_1 \le C_{21} i' + c_2$ (Since j can be arbitrarily large, independent of i and i', it must be that $C_{12}=0$ )

$\rightarrow C_{11} i + c_1 \le C_{21} i' + c_2$ (i <= i')

2. Anti-dependence from read access X[A[i]] to write access X[ j ]

$C_{21} i + c_2 \le [C_{11} \quad C_{12}][\begin{smallmatrix} i' \\ j' \end{smallmatrix}]+c_1$

$\rightarrow C_{21} i + c_2 \le C_{11} i' + C_{12} j' + c_1 (i < i')$

$\rightarrow C_{21} i + c_2 \le C_{11} i' + c_1 (i < i')(since\, C_{12}\text{must be 0 according 1})$

According 1 and 2

$C_{11} i + c_1 \le C_{21} i' + c_2$ (i <= i') → (let i'=i+1) $(C_{21}-C_{11}) i + C_{21} + c_2 - c_1 \ge 0 \ldots (A)$

$C_{21} i + c_2 \le C_{11} i' + c_1 (i < i')$ → (let i'=i+1) $(C_{11}-C_{21}) i + C_{11} + c_1 - c_2 \ge 0 \ldots (B)$

for all i satisfy (A) and (B) *implies*

$\rightarrow C_{11} = C_{21} = 1$

and $C_{12}=0$

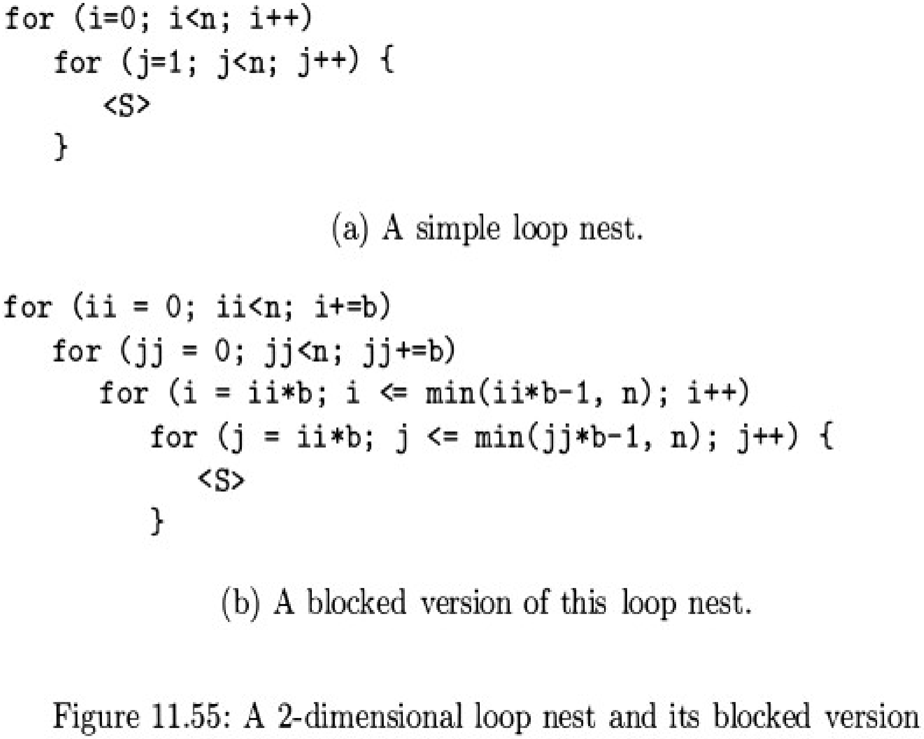
Conclusion, only one independent solution for $[C_{11} \quad C_{12}] \rightarrow$ no time steps mapping

$s_1 : [C_{11} \quad C_{12}][\begin{smallmatrix} i \\ j \end{smallmatrix}] = [1 \quad 0][\begin{smallmatrix} i \\ j \end{smallmatrix}] \rightarrow \textit{original code order}$

# Blocked (Tiling) in k-dimensions

- If there exist k independent solutions to the time-partition constraints of a loop nest, then it is possible has k-deep, fully permutable loop nest.
- A k-deep, fully permutable loop nest can be blocked in k-dimensions.

# Blocked (Tiling) in 2-dimensions



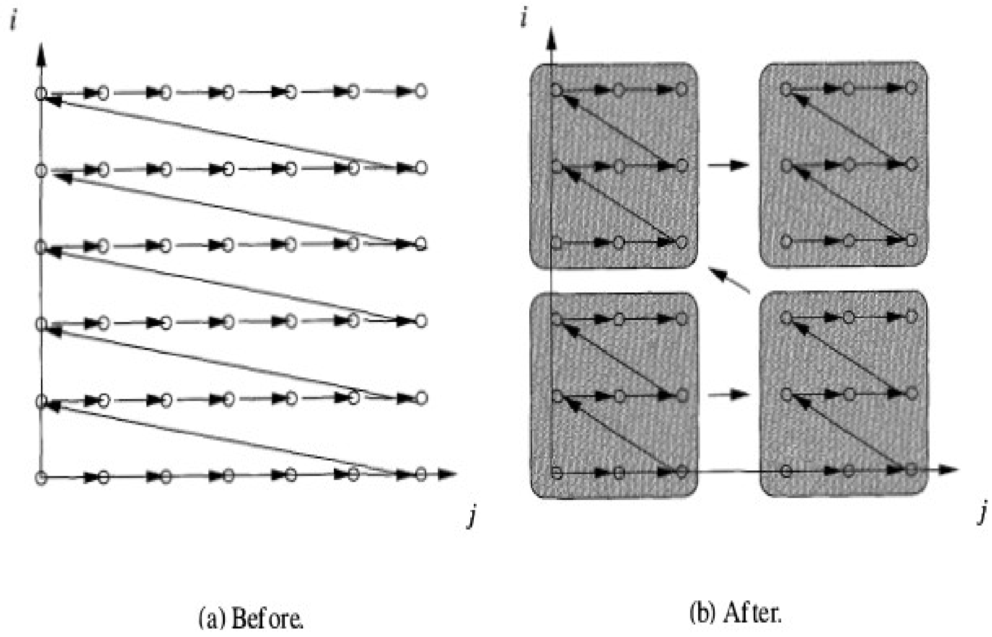
```
for (i=0; i<n; i++)
   for (j=1; j<n; j++) {
      <S>
   }
```

(a) A simple loop nest.

```
for (ii = 0; ii<n; i+=b)
   for (jj = 0; jj<n; jj+=b)
      for (i = ii*b; i <= min(ii*b-1, n); i++)
         for (j = ii*b; j <= min(jj*b-1, n); j++) {
            <S>
         }
```

(b) A blocked version of this loop nest.

Figure 11.55: A 2-dimensional loop nest and its blocked version



(a) Before.

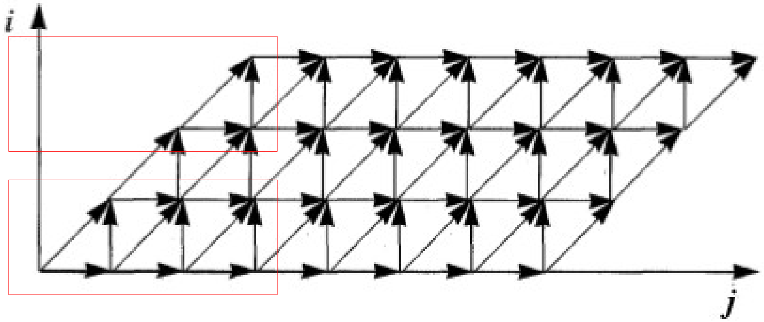
(b) After.

Figure 11.56: Execution order before and after blocking a 2-deep loop nest.

# Blocked (Tiling) in 2-dimensions

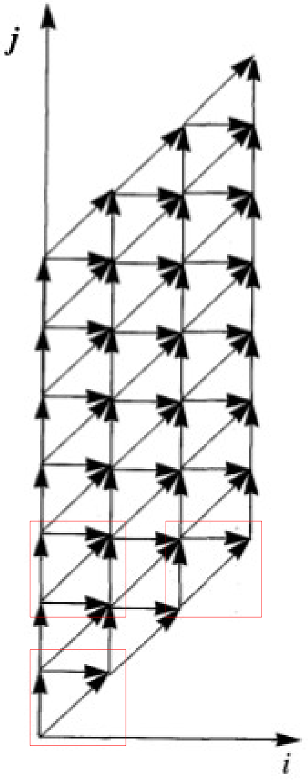
```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])
```

(a) The code in Fig. 11.50 transformed by $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.



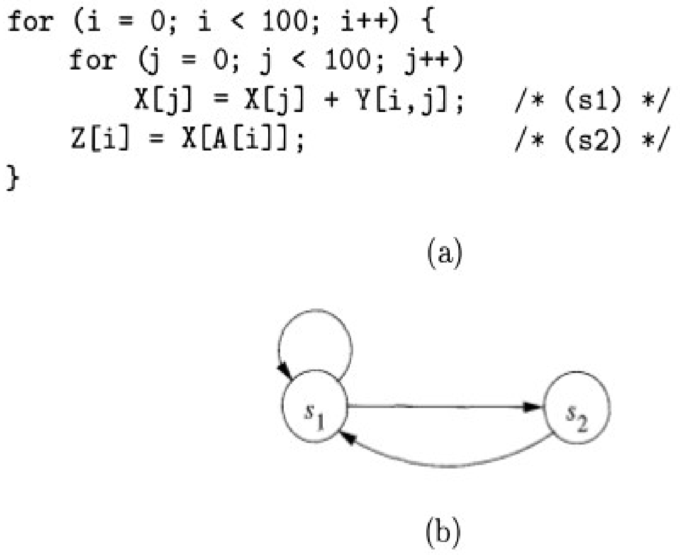
(b) Data dependences of the code in (a).

```
for (j = 0; j <= m+n; i++)
    for (i = max(0,j-n); i <= min(m,j); j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2];
```



(d) Data dependences of the code in (b).

# Blocked (Tiling) cannot in 2-dimensions

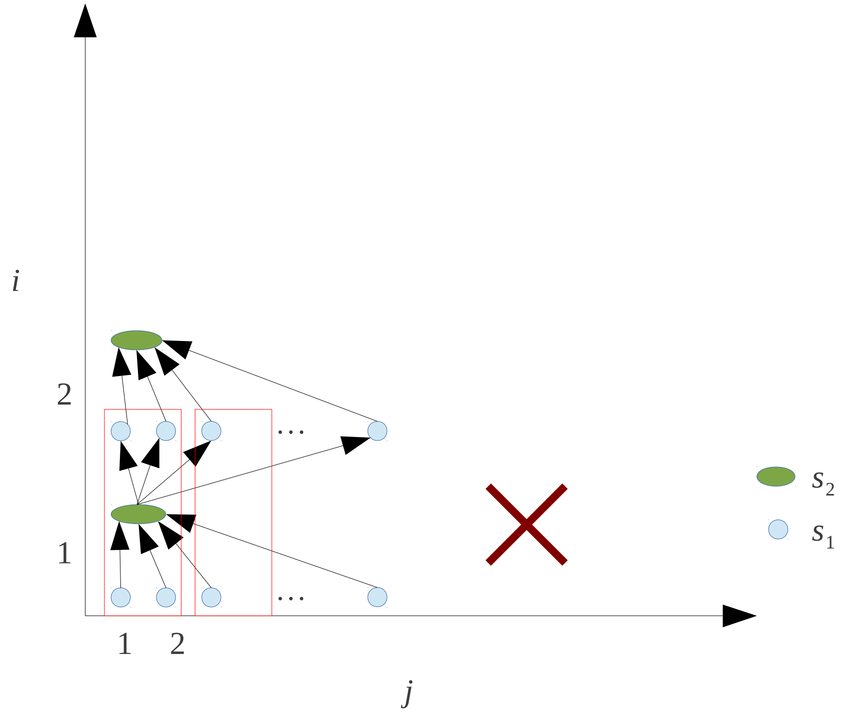


```
for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++)
        X[j] = X[j] + Y[i,j];    /* (s1) */
    Z[i] = X[A[i]];              /* (s2) */
}
```

(a)

(b)

Figure 11.53: A sequential outer loop (a) and its PDG (b)



# Other uses of affine transforms

- Distributed memory machines
- Multi-instruction-issue processors
  - Suppose 4-issues with 2 adders

```
int n=1000;
int m=100;

for ( i = 0 ; i <= n ; i++)
  for ( j = 0 ; j <= n; j++)
    X[i] = X[i] + Y[ j , i ] ;

for ( i = 0 ; i <= m ; i++)
  for ( j = 0 ; j <= n; j++)
    W[i] = W[i] * Z[ j , i ] ;
```

i to inner

```
int n=1000;
int m=100;

for ( j = 0 ; j <= n ; j++)
  for ( i = 0 ; i <= m; i++) {
    ii=i*10;
    X[ii] = X[ii] + Y[ j , ii ] ;
    X[ii+1] = X[ii+1] + Y[ j , ii+1 ] ;
    ...
    X[ii+9] = X[ii+9] + Y[ j , ii+9 ] ;
    W[i] = W[i] * Z[ j , i ] ;
  }
```

1. Spatial locality.
2. Try to make all functional units busy.

# Other uses of affine transforms

- Multi-instruction-issue processors
  - Usage balance

```
int n=1000;

for ( i = 0 ; i <= n ; i++)
  for ( j = 0 ; j <= n; j++)
    X[i] = X[i] + Y[ j , i ] ; // memory bound (since cache miss)

for ( i = 0 ; i <= n ; i++)
  for ( j = 0 ; j <= n; j++)
    W[i] = W[i] / Z[ i , j ] ; // compute bound
```

fusion

```
int n=1000;

for ( i = 0 ; i <= n ; i++)
  for ( j = 0 ; j <= n; j++) {
    X[i] = X[i] + Y[ j , i ] ; // stall
    W[i] = W[i] / Z[ i , j ] ; // balance stall
  }
```

# Other uses of affine transforms

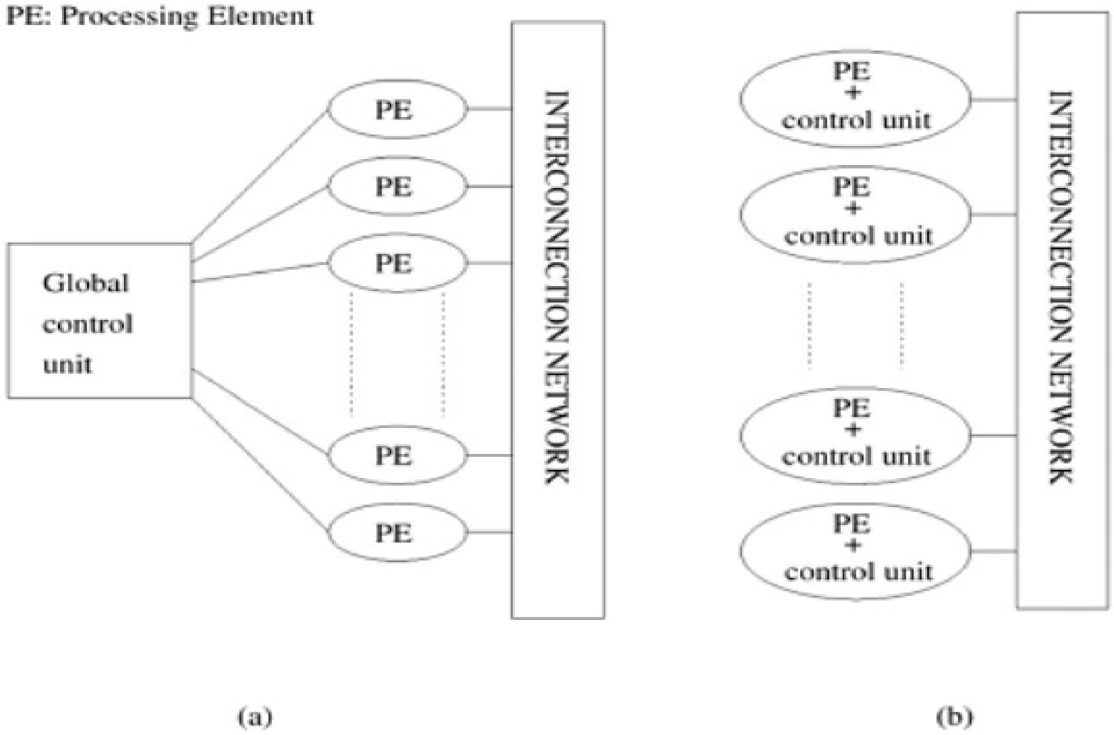
- Vector and SIMD instructions
  - Vector:
    - Elements fetched serial and compute parallel
    - Scatter/gather for not contiguous elements in advanced machine
  - SIMD:
    - Elements fetched parallel, store them in wide registers, and compute parallel
    - 256-byte SIMD operands 256-byte alignment
- Prefetching
  - Issue llvm prefetch IR at best time in polly for marvell cpu.

# Software pipeline

- If it is static schedule machine, then do software pipeline after the pipelining of Polyhedral.
  - Software pipeline will compact instructions cycles via RT table as chapter 10 of DragonBook.

# control

**Figure 2.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).**

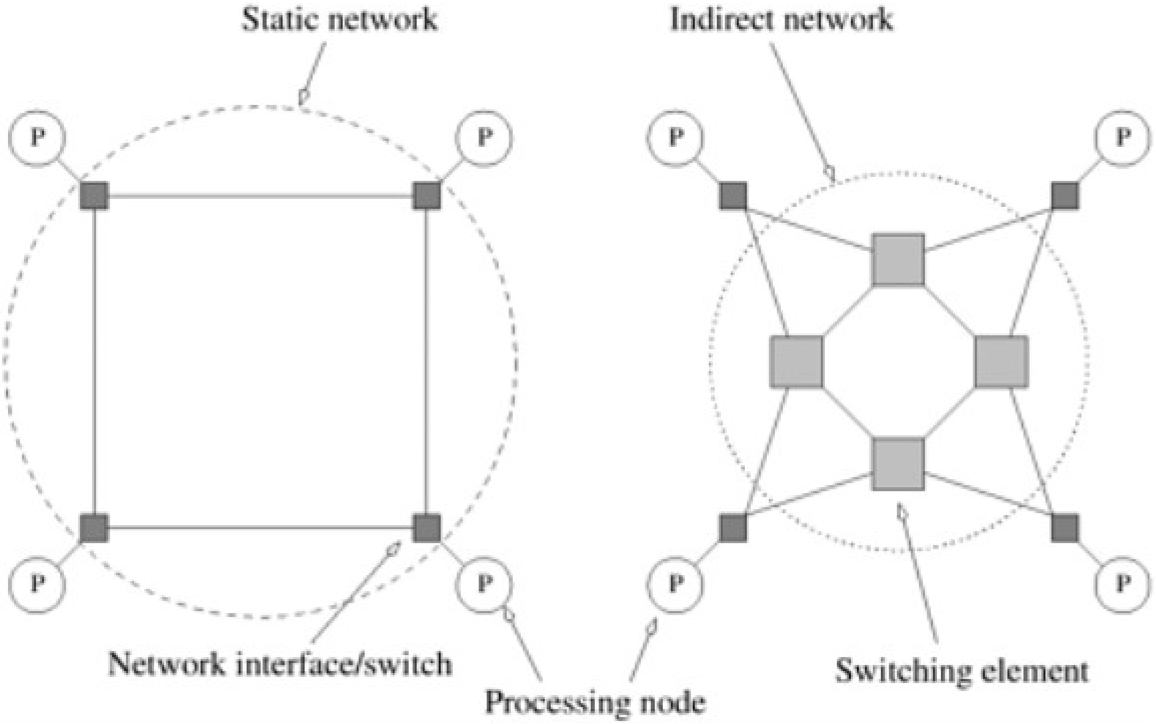


# Interconnection Networks

MIMD can be created by use
Market CPUs + Interconnection Networks

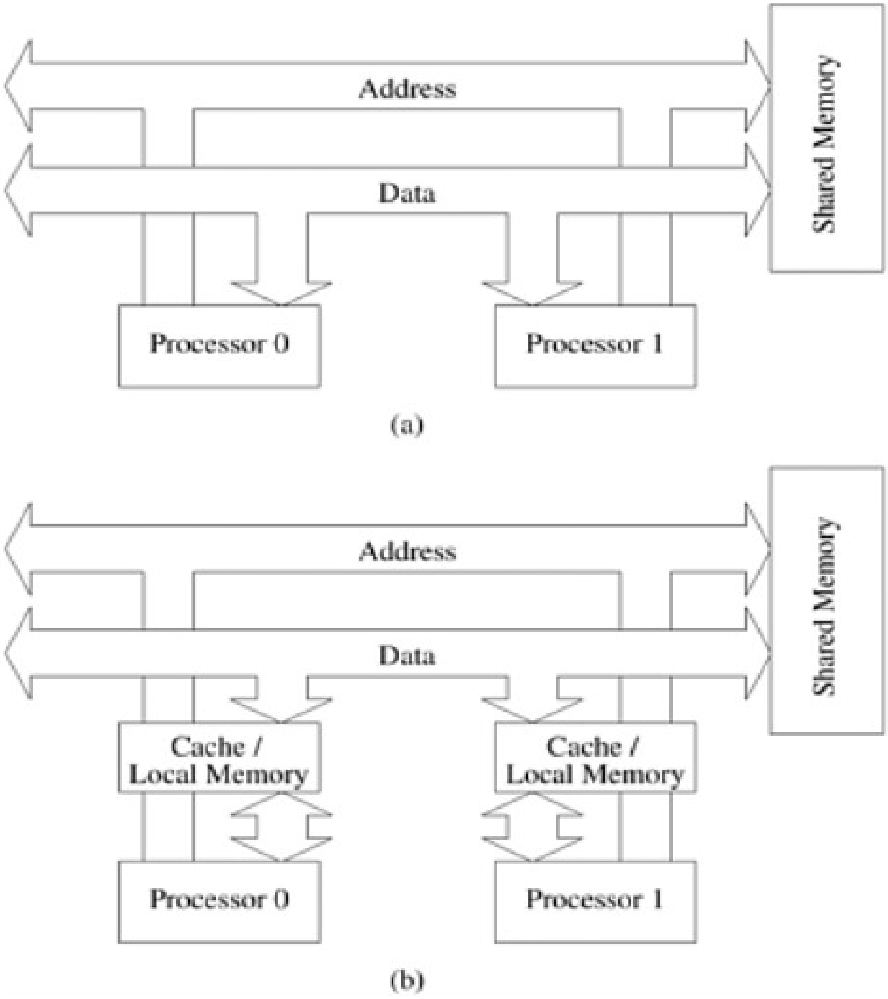
# static v.s. dynamic

**Figure 2.6. Classification of interconnection networks: (a) a static network; and (b) a dynamic network.**



# Dynamic Interconnection Network – Bus

**Figure 2.7. Bus-based interconnects (a) with no local caches; (b) with local memory/ caches.**



# Dynamic Interconnection Network – Bus

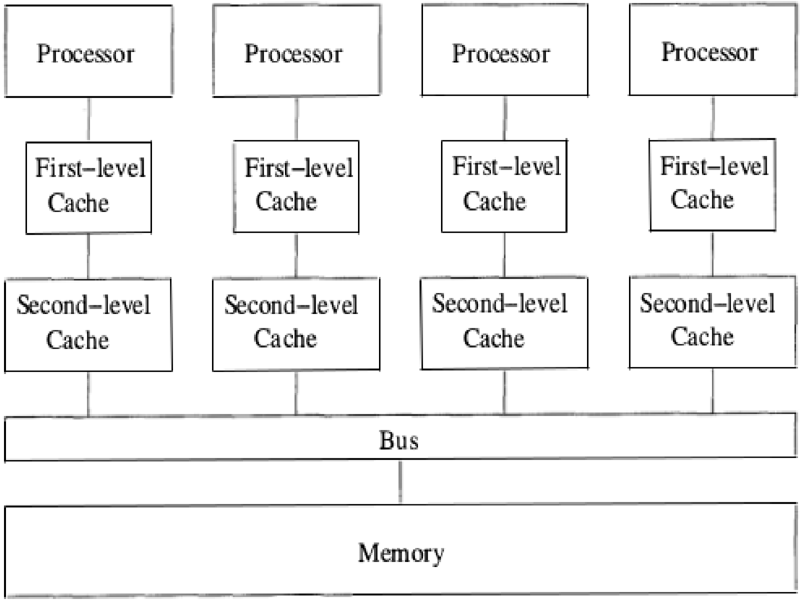


Figure 11.1: The symmetric multi-processor architecture

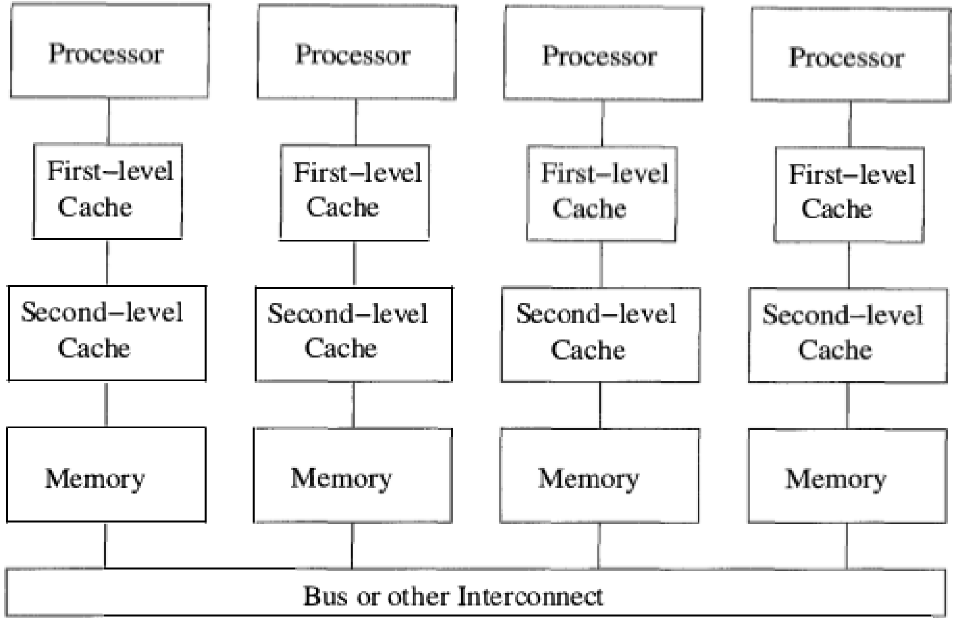
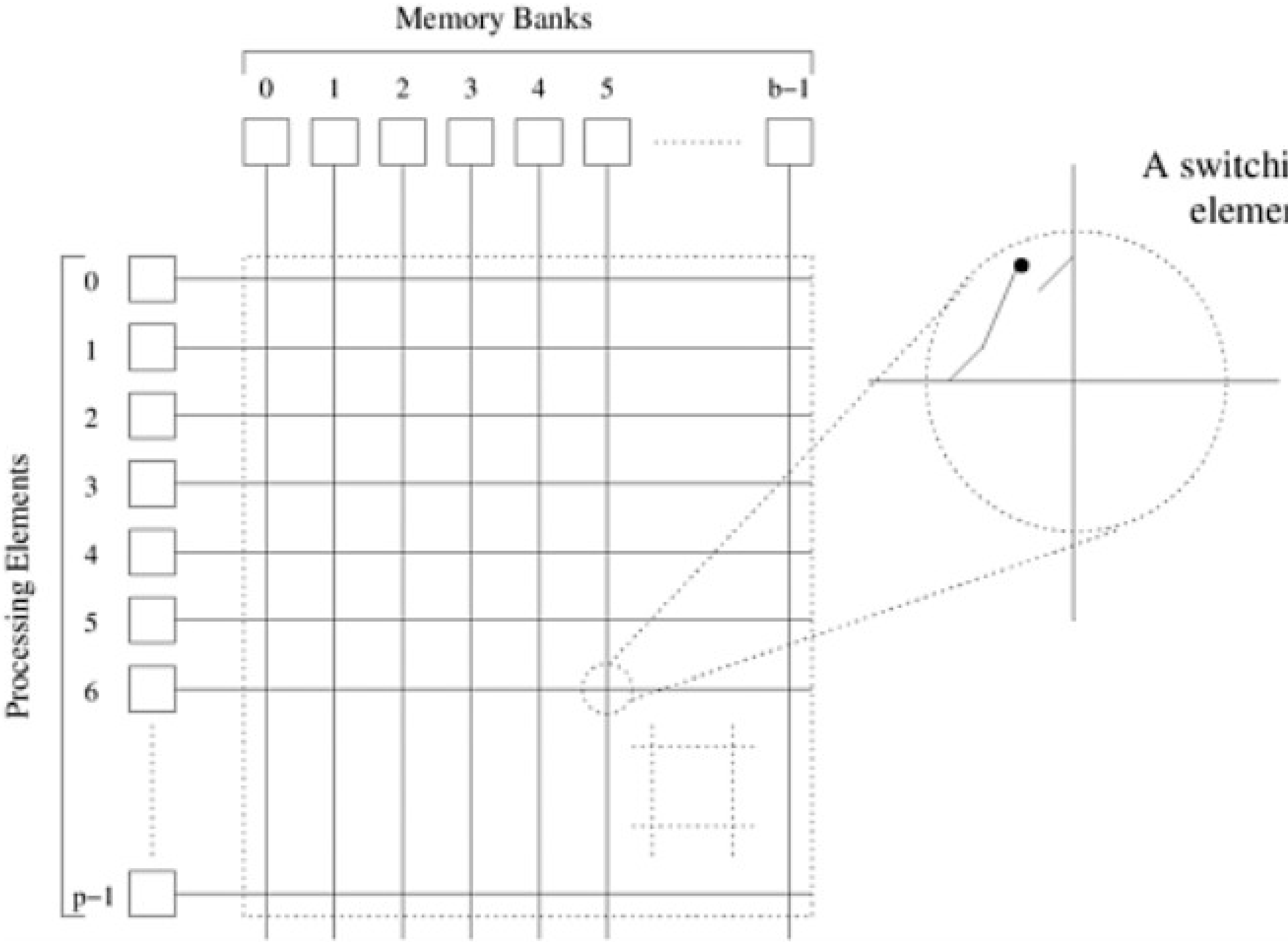


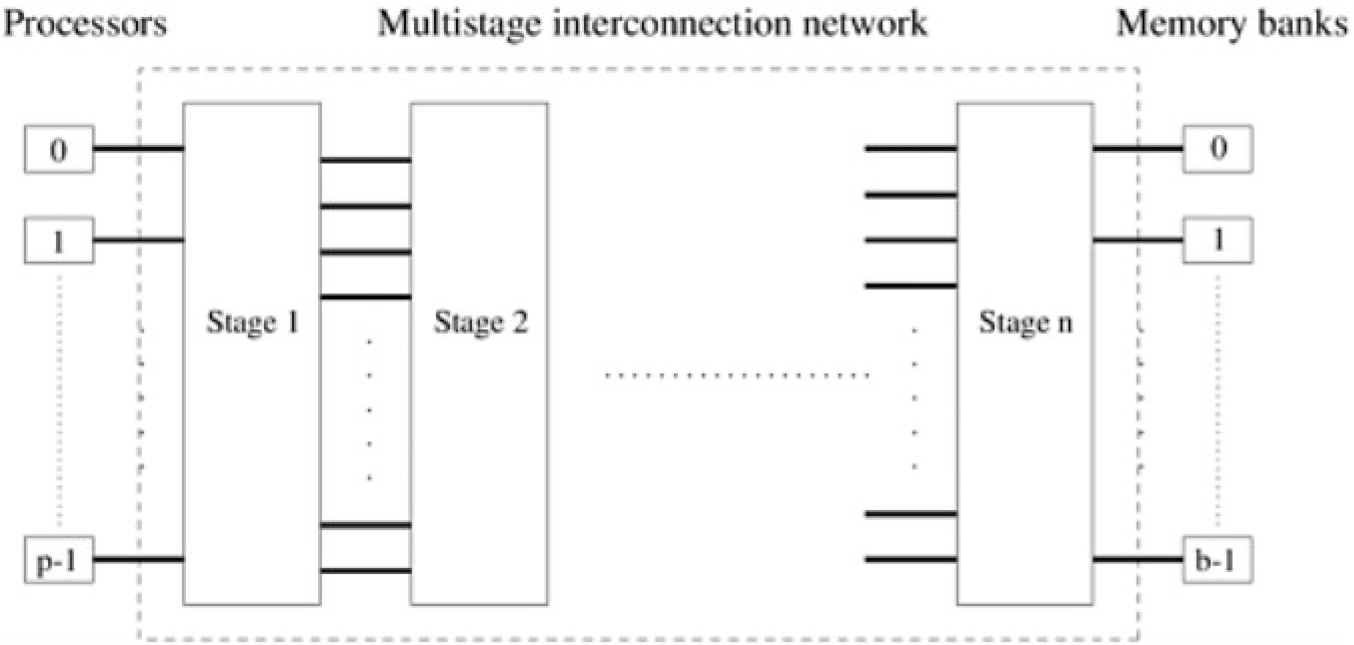
Figure 11.2: Distributed memory machines

# Dynamic Interconnection Network – Crossbar

**Figure 2.8. A completely non-blocking crossbar network connecting p processors to b memory banks.**



# Dynamic Interconnection Network – Multistage

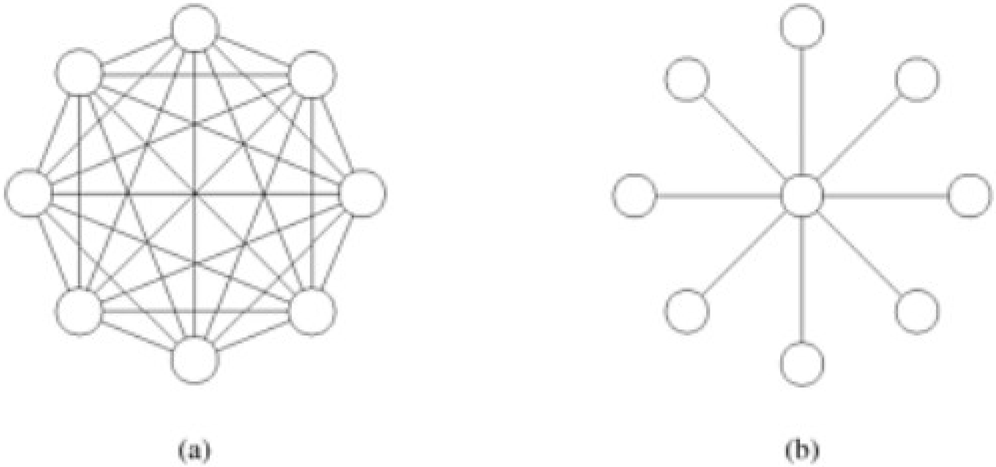



(a)

(b)

# Static Interconnection Network – complete, star

**Figure 2.14. (a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.**

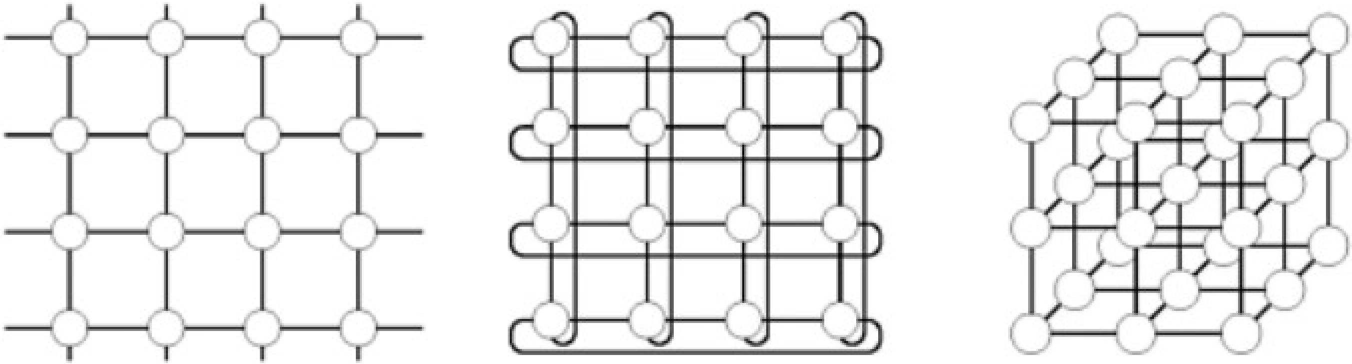


# Static Interconnection Network – ring, mesh

**Figure 2.15. Linear arrays: (a) with no wraparound links; (b) with wraparound link.**

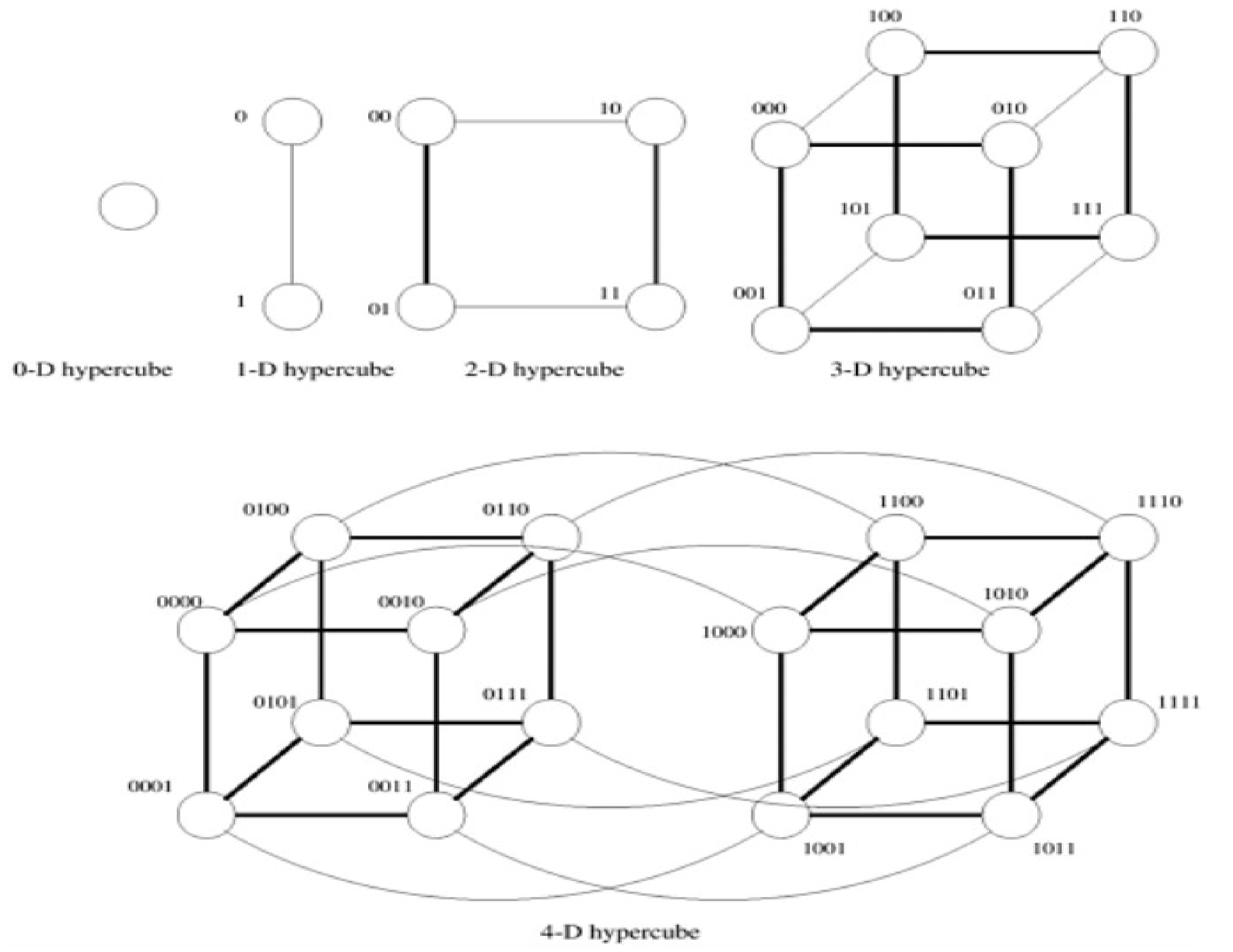

(a) (b)

**Figure 2.16. Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.**


(a) (b) (c)

# Static Interconnection Network – cube

**Figure 2.17. Construction of hypercubes from hypercubes of lower dimension.**



# Tree

Figure 2.18. Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

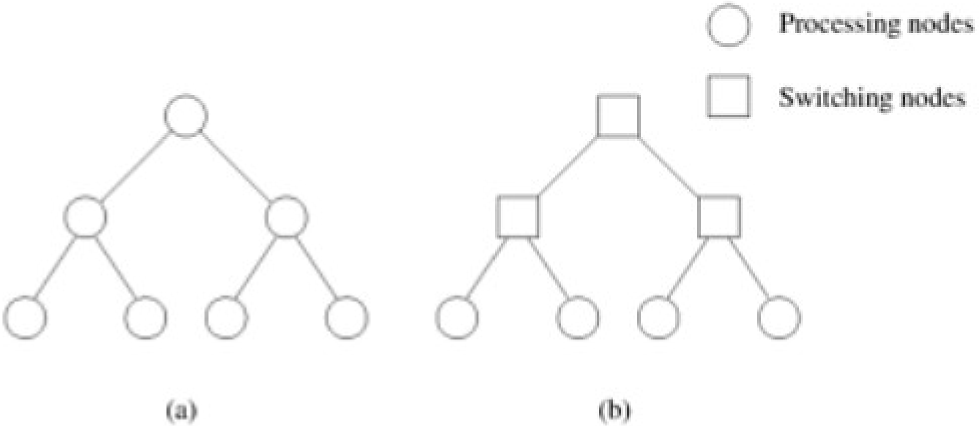
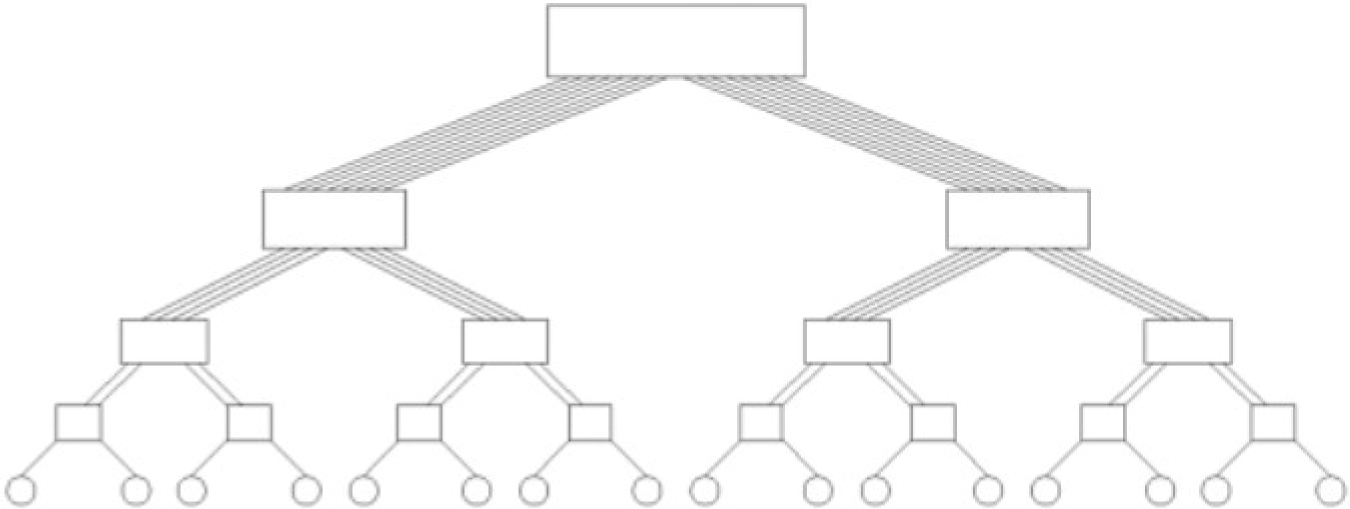


Figure 2.19. A fat tree network of 16 processing nodes.



# Parallel Model Reference

- http://www.top500.org/lists/2013/11/#.U3G8zUl_W5k
- http://www.top500.org/system/177975#.U3CTeEl_W5k
- http://www.cray.com/Products/Computing/XE/Technology.aspx
- http://en.wikipedia.org/wiki/NCUBE

# Reference

- Chapter 11 of Compilers principles, techniques, & tools 2nd edition
- Chapter 2 of Introduction to Parallel Computing, by Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar