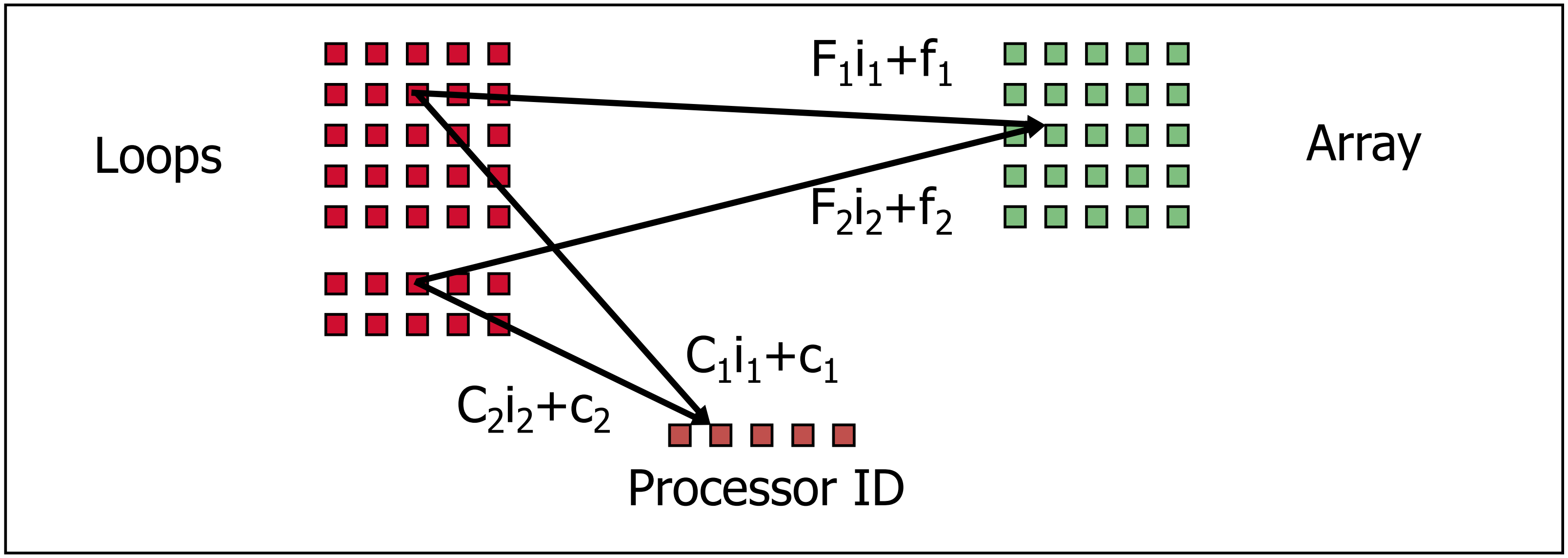
Lecture 11
Pipelined Parallelism

1. Intuition: Time mapping
2. Time Affine Partitioning Problem
3. Time Affine Partitioning Algorithm
4. Coarsest-Grain Parallelization Algorithm
5. Blocking
6. Examples

Readings: Chapter 11.8-11.9

1. Recall: Maximum Parallelism & No Communication



C: Space partitioning of Computation to Processor ID

For every pair of data dependent accesses $F_1i_1+f_1$ and $F_2i_2+f_2$

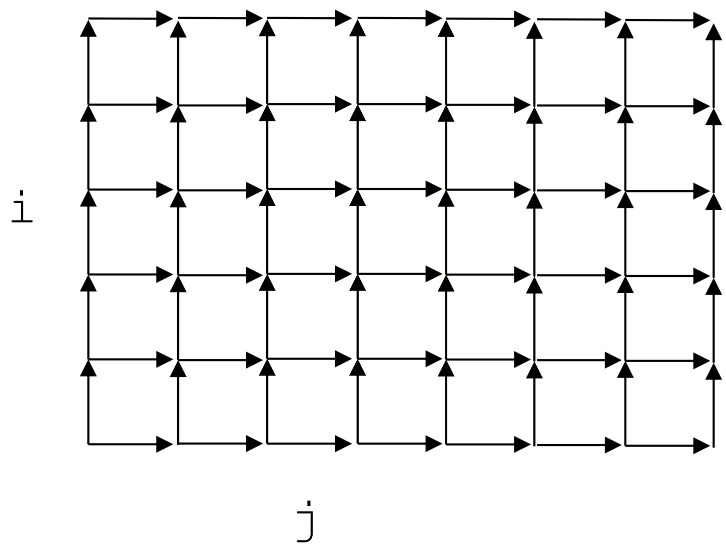
Find C_1, c_1, C_2, c_2:

$$\forall\, i_1, i_2 \quad F_1 i_1 + f_1 = F_2 i_2 + f_2 \rightarrow C_1 i_1 + c_1 = C_2 i_2 + c_2$$

with the objective of maximizing the rank of C_1, C_2

SOR (Successive Over-Relaxation): An Example

```
for i = 1 TO m
   for j = 1 to n
    A[i,j] = c * (A[i-1,j] + A[i,j-1])
```



- Is there communication-free parallelism?
- Can you find parallelism with communication? How?

Pipelineable Parallelism

```
for i = 1 TO m
   for j = 1 to n
      A[i,j] = c * (A[i-1,j] + A[i,j-1])
```

Processor ID: p
Synchronization variable: t[p] initialized to 0
WAIT: thread waits until the condition becomes true

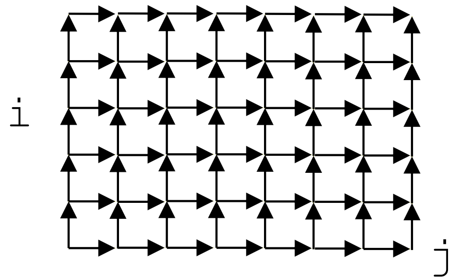
```
   for j = 1 to n
     if (p==1) or (WAIT(t[p-1]>=j))
       A[p,j] = c * (A[p-1,j] + A[p,j-1])
     t[p]++;
```

- Good locality
- Relaxed wavefront
- O(n) synchronization overhead

Sequential Outer Loops

- No communication-free parallelism +> Outer loop must be sequential
- Treat the iteration number of a sequential loop as a partition in "time"

```
for i = 1 TO m
 for j = 1 to n
  A[i,j]=c*(A[i-1,j]+ A[i,j-1])
```



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

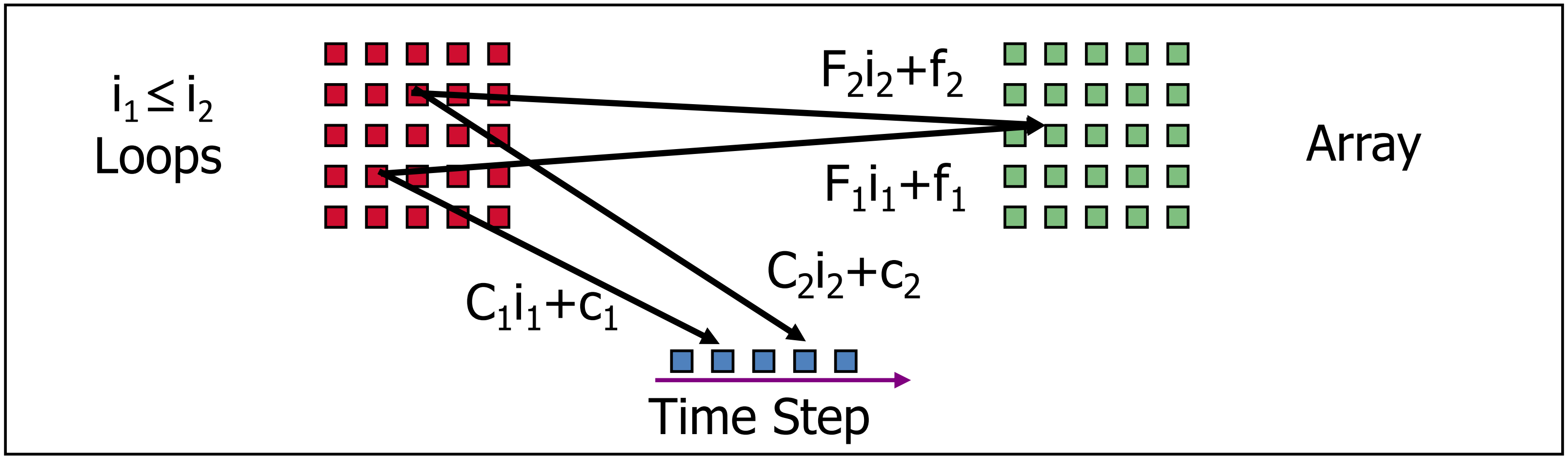
- Is j also a legal outer loop?
- Two independent basis vectors: $[1 \quad 0]$, $[0 \quad 1]$
  - Any combination of $[1 \quad 0]$, $[0 \quad 1]$ is a legal time partitioning
  - All original data dependences do not point backward in time
- **Choice means parallelism**
- The solution space has rank 2
  - 1 degree of pipelined parallelism

r-Dimensional Pipelineable Parallelism

- r-dimensions of legal time mapping:
  - r-1 degrees of parallelism
    - $O(n^{r-1})$ parallelism
    - $O(n)$ synchronization

- Synchronization
  - processor ID $(p_1, p_2, \ldots, p_{r-1})$:
    - r-1 outer loops map to each processor
    - Runs rth loop sequentially on each processor
  - iteration i_r for processor $(p_1, p_2, \ldots, p_{r-1})$, waits for iteration i_r for processors $(p_1-1, p_2, \ldots, p_{r-1})$, $(p_1, p_2-1, \ldots, p_{r-1})$, …, $(p_1, p_2, \ldots, p_{r-1}-1)$.

2. Finding Maximum Pipelinable Parallelism



C: Time Partitioning of Computation to Time Step

For every pair of data dependent accesses $F_1 i_1 + f_1$ and $F_2 i_2 + f_2$

Let $B_1 i_1 + b_1 \geq 0$, $B_2 i_2 + b_2 \geq 0$ be the corresponding loop bound constraints,

Find C_1, c_1, C_2, c_2:

$$\forall\, i_1, i_2 \quad B_1 i_1 + b_1 \geq 0,\ B_2 i_2 + b_2 \geq 0$$

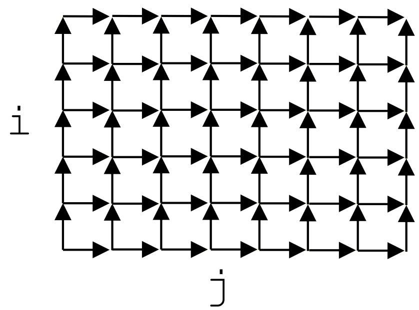
$$(i_1 \leq i_2) \wedge (F_1 i_1 + f_1 = F_2 i_2 + f_2) \rightarrow C_1 i_1 + c_1 \leq C_2 i_2 + c_2$$

with the objective of maximizing the rank of C_1, C_2

Example 1

(a)

```
for i = 1 TO m
 for j = 1 to n
  A[i,j]=c*(A[i-1,j]+ A[i,j-1])
```



2 time mappings:

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

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

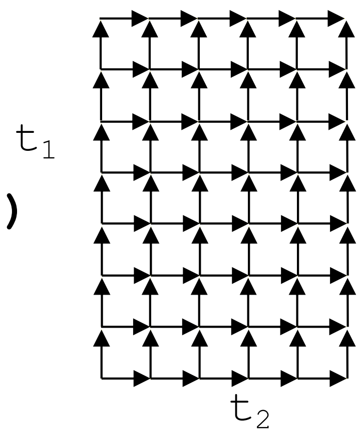
2 legal permutations

(a) $\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}$

(b) $\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}$

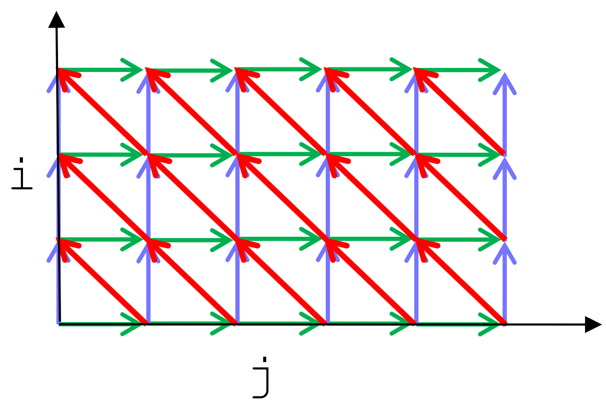
(b)

```
for t1 = 1 TO n
 for t2 = 1 to m
  A[i,j]=c*(A[i-1,j] + A[i,j-1])
```



Example 2

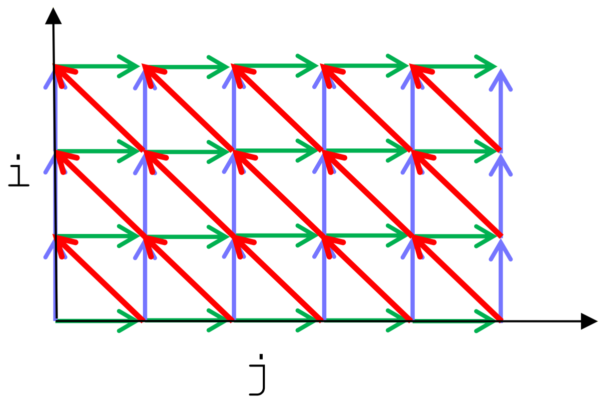
```
for i = 0 TO m
 for j = 0 to n
  X[j+1]=(X[j]+X[j+1]+X[j+2])/3
```



Is the above loop permutable as is?
(Can we make both i and j outer loops?)

Time Partitioning Results

```
for i = 0 TO m
 for j = 0 to n
  X[j+1]=(X[j]+X[j+1]+X[j+2])/3
```



2 time mappings:

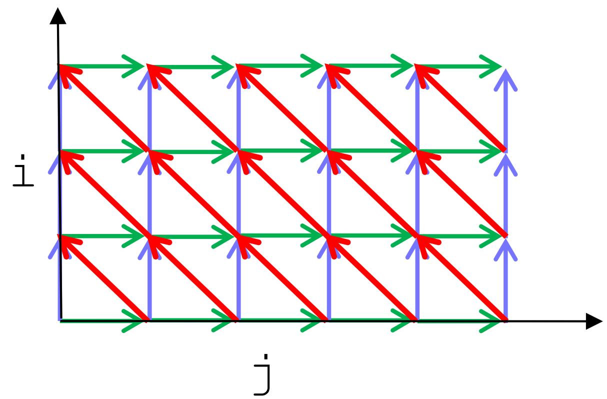
$$[t] = [1 \quad 0]\begin{bmatrix} i \\ j \end{bmatrix} + [0]$$

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

Intuitively:
The time mapping makes all
the dependences not point backwards.

Time Partitioning Results

```
for i = 0 TO m
  for j = 0 to n
    X[j+1]=(X[j]+X[j+1]+X[j+2])/3
```



2 time mappings:

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

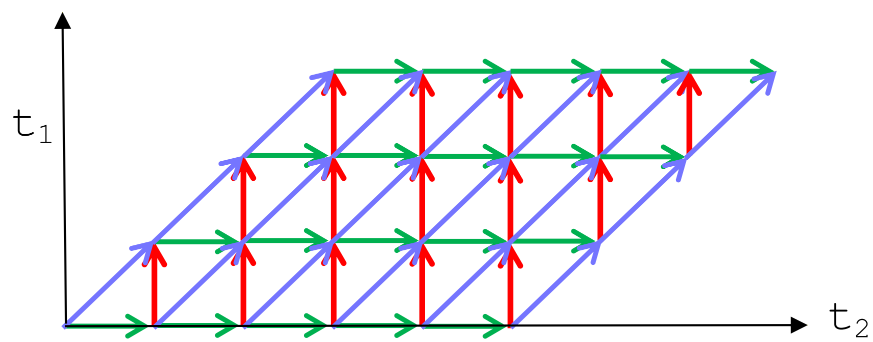
$$[t] = [1 \quad 1]\begin{bmatrix} i \\ j \end{bmatrix} + [0]$$

2 permutations:

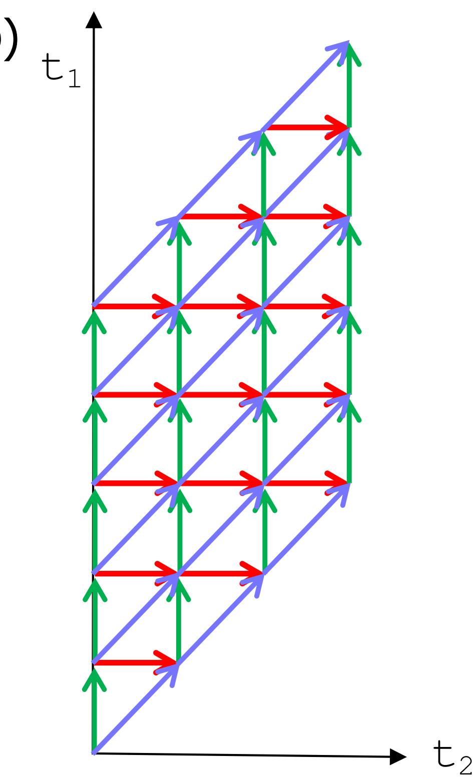
(a) $\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}$

(b) $\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}$

(a)



(b)



Fully Permutable Loop Nests

- Definition:
  A loop nest is <u>fully permutable</u>
  if all the loops in the nest can be permuted arbitrarily
  without changing the semantics of the program

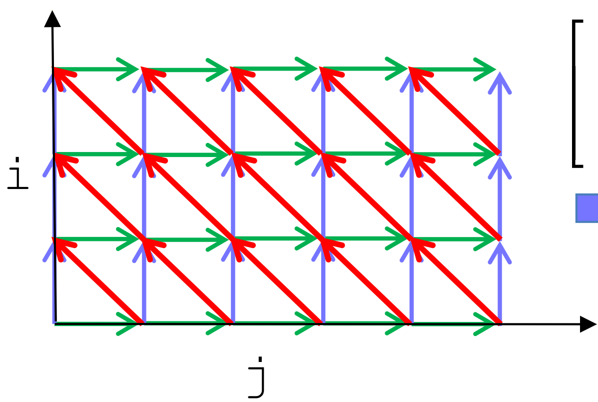
- Affine time partitioning algorithm finds fully permutable loop nests.

- Rank r time mappings
  - r possible outermost loops
  - Dependences do not point backward along r-axes
  - Rank r matrix (combining all time mappings)
    transforms the original loop
    into r-deep outermost fully permutable nest

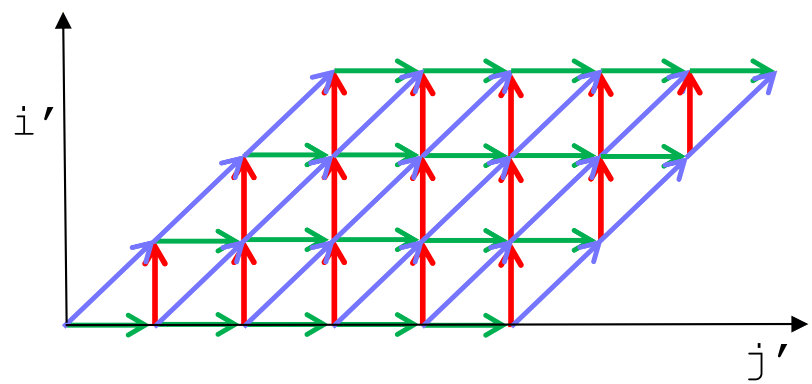
Code Generation

```
for i = 0 TO m
 for j = 0 to n
  X[j+1]=(X[j]+X[j+1]+X[j+2])/3
```

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



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



Transformation

i' = i

j' = i + j

j = j' – i'

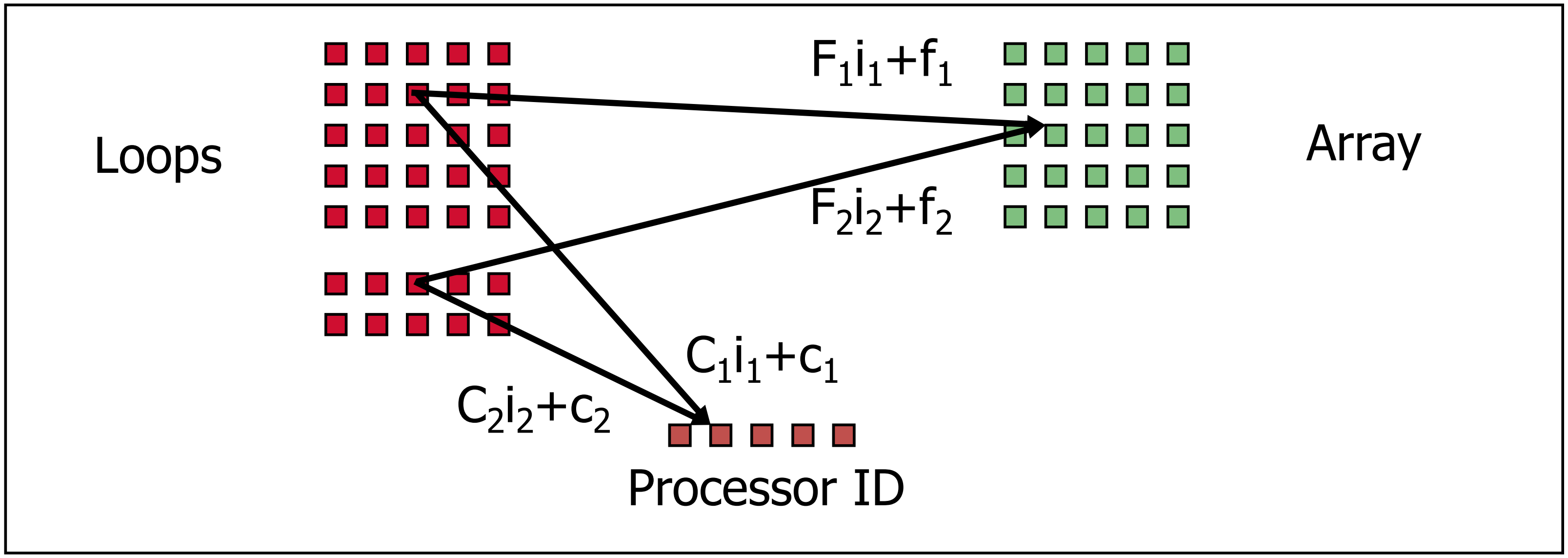
Loop bounds (Using Fourier-Motzkin Elimination)

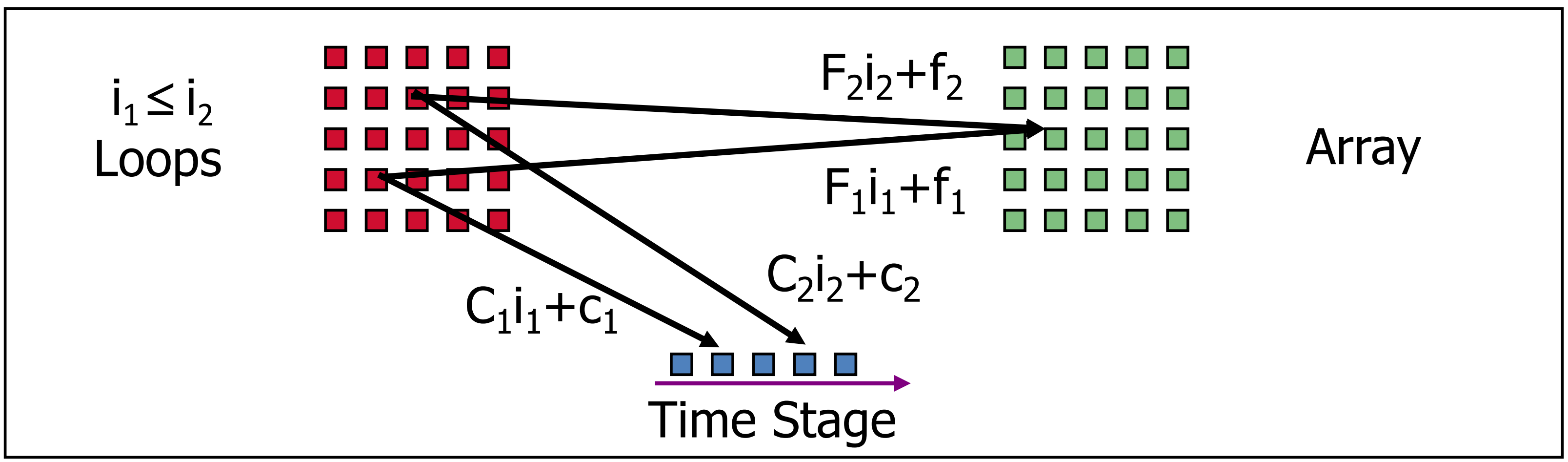
0 <= i' <=m

0 <= j'- i' <= n

3. Time Partitioning Algorithm

Compare:





Comparing the Two Problems

Communication-Free Parallelism:

C: Space partitioning of Computation to Processor ID

For every pair of data dependent accesses $F_1 i_1 + f_1$ and $F_2 i_2 + f_2$

Find C_1, c_1, C_2, c_2:

$$\forall\ i_1, i_2 \quad F_1 i_1 + f_1 = F_2 i_2 + f_2 \rightarrow C_1 i_1 + c_1 = C_2 i_2 + c_2$$

with the objective of maximizing the rank of C_1, C_2

Pipelining Parallelism:

C: Time mapping of Computation to Time

For every pair of data dependent accesses $F_1 i_1 + f_1$ and $F_2 i_2 + f_2$

Let $B_1 i_1 + b_1 \geq 0$, $B_2 i_2 + b_2 \geq 0$ be the corresponding loop bound constraints,

Find C_1, c_1, C_2, c_2:

$$\forall\ i_1, i_2 \quad B_1 i_1 + b_1 \geq 0,\ B_2 i_2 + b_2 \geq 0$$

$$(i_1 \leq i_2) \wedge (F_1 i_1 + f_1 = F_2 i_2 + f_2) \rightarrow C_1 i_1 + c_1 \leq C_2 i_2 + c_2$$

with the objective of maximizing the rank of C_1, C_2

Farkas Lemma

Finding the possible time dimensions c:

Given matrix A, find a vector c such that

for all vectors x such that $Ax \geq 0$,

$c^T x \geq 0$

Farkas Lemma, 1901 (real domain)

The primal system of inequalities

$Ax \geq 0, \quad c^T x < 0$

has a real-valued solution x

or, the dual system

$A^T y = c, \quad y \geq 0$

has a real-valued solution y, but never both.

Time partitioning: Find c such that $A^T y = c, \quad y \geq 0$

Note: Farkas Lemma: a theorem of the alternative (no intuitive proof exists)

4. O(1) Synchronization

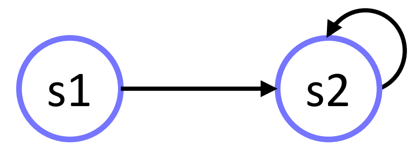
What if there is only 1 legal outermost loop?

Example:

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

O(1) Synchronization Algorithm

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




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

- Program dependence graph
  - Nodes: statements
  - Edges: data dependence
- Algorithm
  - Split the program into
    a sequence of strongly connected components
    separated by O(1) barriers
  - Find communication-free parallelism in components
    (if no component has communication-free parallelism,
    leave as one partition).

Coarsest Parallelism with Minimum Synchronization

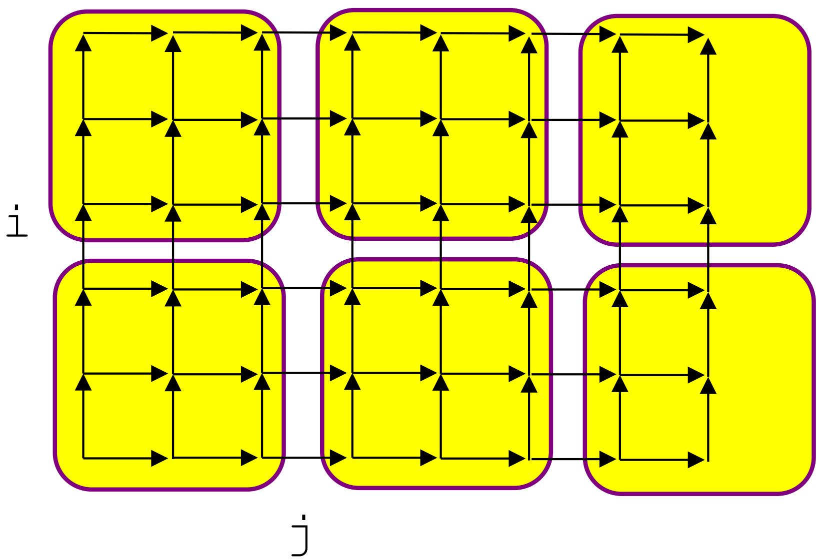
Find parallelism with coarsest parallelism with minimum synchronization

a. Find outermost communication-free parallelism
b. Find O(1) synch. parallelism to partitions* found.
c. For each partition found,
   Find outermost fully permutable loop nest O(n) synch
   Find O(1) synch. parallelism on partitions* found.
d. Recursively apply c to inner loops if any.

*partition = code for each processor (if parallelism is found)
full code otherwise

5. Blocking

```
for (i = 1; i < m; i++) {
   for (j = 1; j < n; j++) {
      A[i,j] = c * (A[i-1,j] + A[i,j-1]) }}
```



Reduces synchronization
Increases locality
- minimize communication
- improve cache performance
- (applicable to registers as well!)

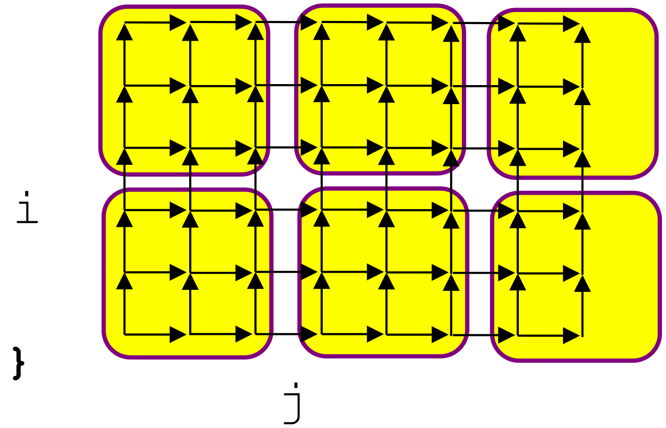
How to Block Loops?

- Fully permutable loop nests can be blocked
- Let B be the block size
  1. Stripmine a loop into 2
     (This is legal for all loops)
     ```
     for (i = 0; i < n; i++) {
       <code>
     }
     =>
     for (ii = 0; ii < n; ii = ii+B) {
        for (i = ii; i < min(n,ii+B); i++) {
             <code>
     }}
     ```

  2. If fully permutable: permute inner stripmined loop inside

Example:	n = 7; B = 3
	i = 0 1 2 3 4 5 6 7
ii = 0	i = 0 1 2
ii = 3	i = 3 4 5
ii = 6	i = 6 7

Blocking SOR



For simplicity, assume m = n = 101; B = 10

- Original program

```
for (i = 1; i < 101; i++) {
  for (j = 1; j < 101; j++) {
    A[i,j] = c * (A[i-1,j] + A[i,j-1]) }}}
```

- Stripmine loops

```
for (ii = 1; ii < 101; ii = ii+10) {
  for (i = ii; i < ii+10; i++) {
    for (jj = 1; jj < 101; jj = jj+10) {
      for (j = jj; j < jj+10; j++) {
        A[i,j] = c * (A[i-1,j] + A[i,j-1]) }}}
     }}}
```

- Permute loops

```
for (ii = 1; ii < 101; ii = ii+10) {
   for (jj = 1; jj < 101; jj = jj+10) {
     for (i = ii; i < ii+10); i++) {
        for (j = jj; j < jj+10; j++) {
           A[i,j] = c * (A[i-1,j] + A[i,j-1]) }}}
}}}
```

Another Use of Blocking

- Coarse-grain parallel loops can be blocked
  to improve parallelism or locality in inner loops.

  - Example: multiprocessor, with instruction-level parallelism

```
for (i = 0; i < n; i++) {
   for (j = 1; j < n; j++) {
        Z[i,j] = Z[i,j-1]
 }}

=> for (ii = 0; ii < n; ii+=16) {
   for (j = 0; j < n; j++) {
     for (i = ii; ii < min(n, ii+16); i++) {
      Z[i,j] = Z[i,j-1];
   }}}
```

6. Example: Neural Network

```
// 2D 3x3 convolution (stride=1)
for i = 0 to channels-1
  for y = 2 to Sy-1
    for x = 2 to Sx-1
      B[i,y,x] = A[i,y-2,x-2]*W1[0,0]+ A[i,y-2,x-1]*W1[0,1]+…
                 A[i,y-1,x-2]*W1[1,0] +…
                 A[i,y,x-2]*W1[2,0]    +…

// ReLU (Rectified Linear Unit)
for i = 0 to channels-1
  for y = 2 to Sy-1
    for x = 2 to Sx-1
      B[i,y,x] = max(B[i,y,x], 0)

// 2D 3x3 convolution (Stride = 2)
for i = 0 to channels-1
  for y = 2 to (Sy-1)/2
    for x = 2 to (Sx-1)/2
      C[i,y,x] = B[i,2*y-2,2*x-2]*W2[0,0] + …
                 B[i,2*y-1,2*x-2]*W2[1,0] + …
```

```
// Dense neural network layer
for i = 0 to channels-1
  for j = 0 to Sj-1
    for y = 2 to (Sy-1)/2
      for x = 2 to (Sx-1)/2
        D[i,j] += C[i,y,x]*W3[j,y,x]

// Softmax:
for i = 0 to channels-1
  for j = 0 to Sj-1
    T[i,j] = exp(D[i,j]);
    E[i] += T[i,j];

  for j = 0 to Sj-1
    F[i,j] = T[i,j]/E[i]
```

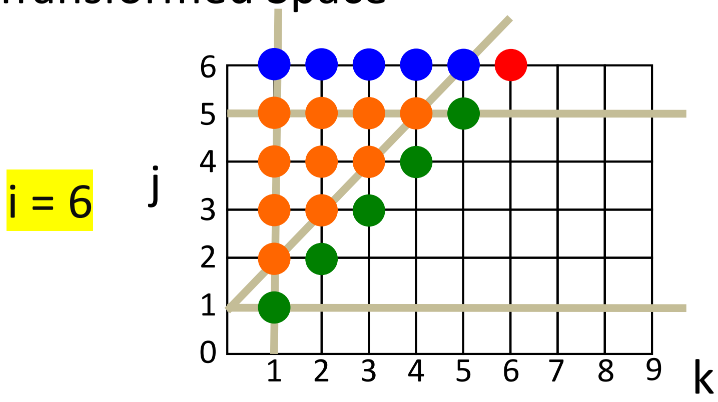
Parallelization without Reduction Optimization

```
// 2D convolution (stride=1)
for i = 0 to channels-1        // Parallel loop
  for y = 2 to Sy-1            // Permutable loop nest
    for x = 2 to Sx-1          // Permutable loop nest
      // 2D convolution
      B[i,y,x] + = A[i,y-2,x-2]*W1[0,0]+ A[i,y-2,x-1]*W1[0,1]+...
                   A[i,y-1,x-2]*W1[1,0] +...
                A[i,y,x-2]*W1[2,0]    +...
      // ReLU (Rectified Linear Unit)
      B[i,y,x] = max(B[i,y,x], 0)

      // 2D convolution (Stride = 2)
      if (y >=4) && (x >=4) && (y mod 2 == 0) && (x mod 2 == 0)
        C[i,y/2,x/2] += B[i,y-2,x-2]*W2[0,0] + ...
                        B[i,y-1,x-2]*W2[1,0] + ...

  // Dense neural network layer
  for j = 0 to Sj-1          /* Parallel loop */
    for y = 2 to (Sy-1)/2
      for x = 2 to (Sx-1)/2
        D[i,j] += C[i,y,x]*W3[j,y,x]
    T[i,j] = exp(D[i,j]);

  // Softmax
  for j = 0 to Sj-1
    E[i] += T[i,j];
  for j = 0 to Sj-1        /* Parallel loop */
    F[i,j] = T[i,j]/E[i]
```

Example: Cholesky Decomposition

```
for (i = 1; i <= N; i++) {
  for (j = 1; j <= i-1; j++) {
    for (k = 1; k <= j-1; k++)
      X[i,j] = X[i,j] - X[i,k]*X[j,k];
    X[i,j] = X[i,j]/X[j,j];  }
  for (m=1; m<=i-1; m++) {
    X[i,i]=X[i,i]-X[i,m]*X[i,m];}
  X[i,i] = sqrt(X[i,i]); }
```

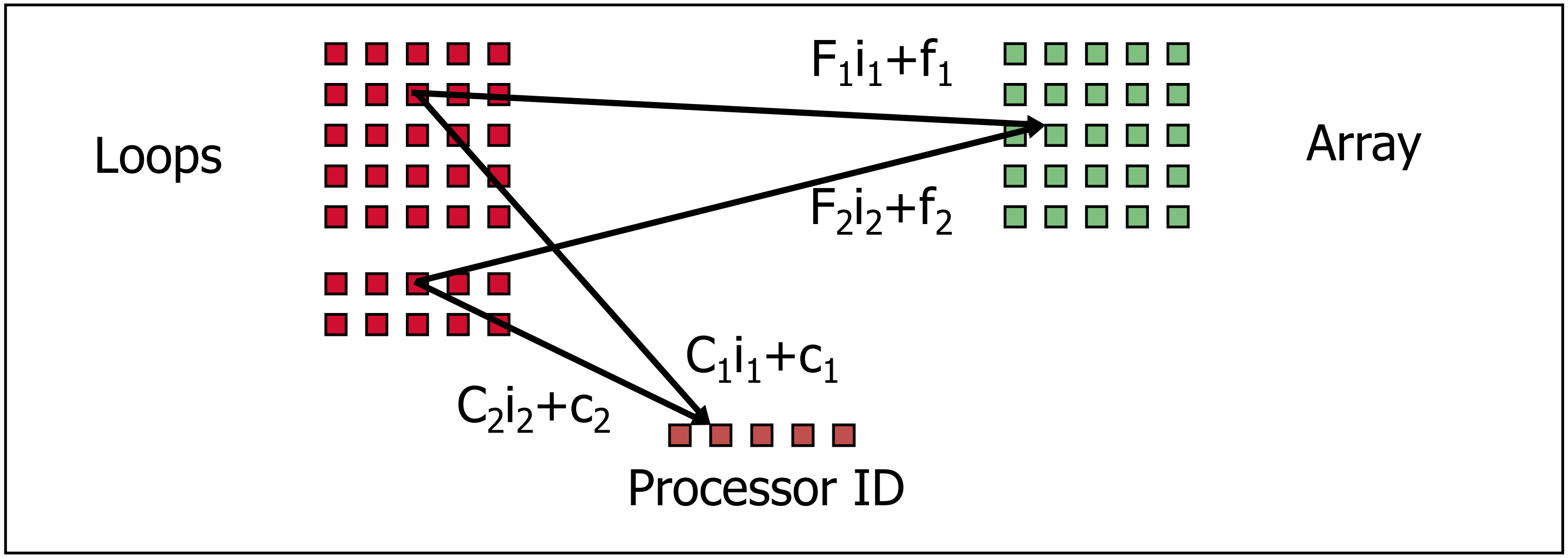
Transformed Space

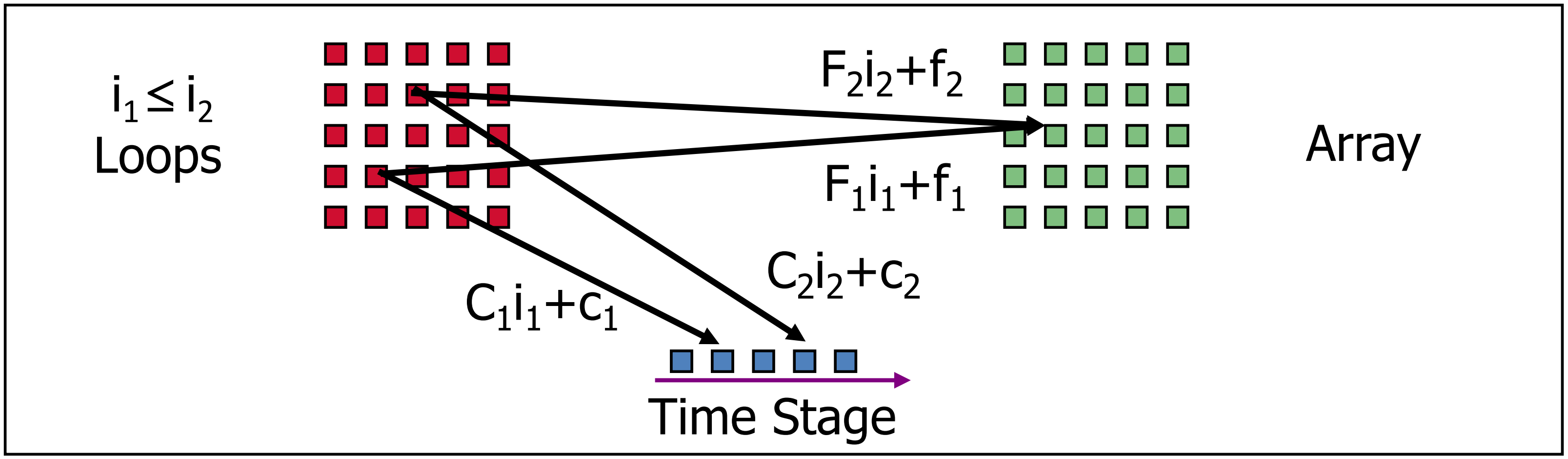



```
for (i = 1; i <= N; i++) {
  for (j = 1; j <= i; j++) {
    for (k = 1; k <= i; k++)
      if (j<i && k<j)
        X[i,j] = X[i,j] - X[i,k]*X[j,k];
      if (j==k && j<i)
        X[i,j] = X[i,j]/X[j,j];
      if (i==j && k<i)
        X[i,i]=X[i,i]-X[i,k]*X[i,k];
      if (i==j && j==k)
        X[i,i] = sqrt(X[i,i]); }}}
```

3-deep fully permutable loop nest
2-dimensional parallelism

Summary: Two Key Algorithms





General Lessons

- Elegant mathematical approach
  - Exploit regularity in affine array accesses with affine mappings
  - Better performance, easier to get a correct compiler
- Coarse-grain parallelism: canonical representation of parallelism
  - Can block to tailor the code for specific machine architecture:
    - instruction-level parallelism, SIMD operations, cache/register locality
- Compiler advantage: Portability across machine models