Lecture 12
Parallelism & Locality

1. Coarsest Granularity of Parallelsim
2. Stripmining: SIMD, Instruction Level Parallelism
3. Blocking for Locality
4. Real-world examples

Readings: Chapter 11.8-11.9

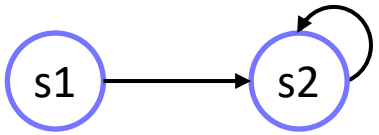
Finding Outermost Parallelism

- Affine transforms
  - Outermost communication-free parallelism (DoAll Loops)
  - Outermost permutable loop nest (Pipelinable loops)
  - Note: DoAll loops are trivially permutable loops

- What if there is only 1 outermost fully permutable loop?

O(1) Synchronization Algorithm (single loop barrier)

```
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).

Algorithm to Find the Coarsest-Grain 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 split parallelizable code

c. For each parallelizable code unit found,

    Find outermost fully permutable loop nest O(n) synch
    Find O(1) synch. parallelism to split parallelizable code

d. Recursively apply c to inner loops if any.

2. Stripmining

- Stripmining: turns a single loop into a nesting of 2 loops

- Let B be the block size,

```
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>
}}
```

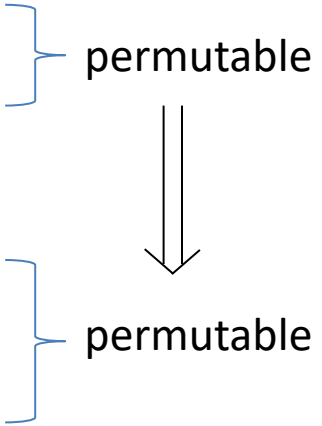
Example: n = 7; B = 3
Introduce ii = 0 3 6
i = 0 1 2 3 4 5 6 7

- Stripmining is legal for any loop (no change to the execution)

Stripmining + Permutation

- Given a fully permutable loop nest
  - stripmined loop can be permuted inwards

```
    for (i = 0; i < n; i++) {
      for (j = 0; j < m, j++) {
         <code>
    }
=>
    for (ii = 0; ii < n; ii = ii+B) {
       for (i = ii; i < min(n,ii+B); i++) {
          for (j = 0; j < m, j++) {
             <code>
    }}}
=>
    for (ii = 0; ii < n; ii = ii+B) {
       for (j = 0; j < m, j++) {
          for (i = ii; i < min(n,ii+B); i++) {
             <code>
    }}}
```

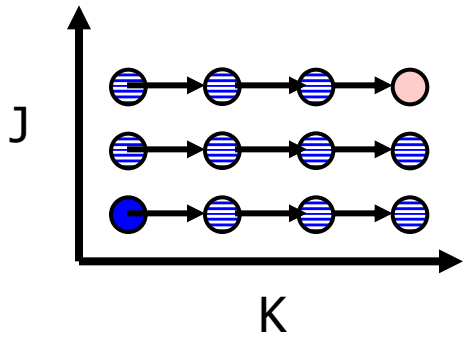


Parallel Architectures

- Multiprocessors often have processors with
  - Instruction level parallelism
    - Software pipelining needs parallelism in the innermost loop
  - SIMD instructions
    - Needs consecutive data access in innermost loop
    - E.g. **`C[i:i+4] = A[i:i+4] + B[i:i+4]`**

Multiprocessor with Instruction Level Parallelism

```
for J = 0; J < N; J++
  for K = 1; K < N; K++
    Z[K,J] = Z[K-1,J]
```




```
for JJ = 0; JJ < N; JJ+=B
  for K = 1; K < N; K++
    for J = JJ; J < min(N, J+B); J++
      Z[K,J] = Z[K-1,J]
```

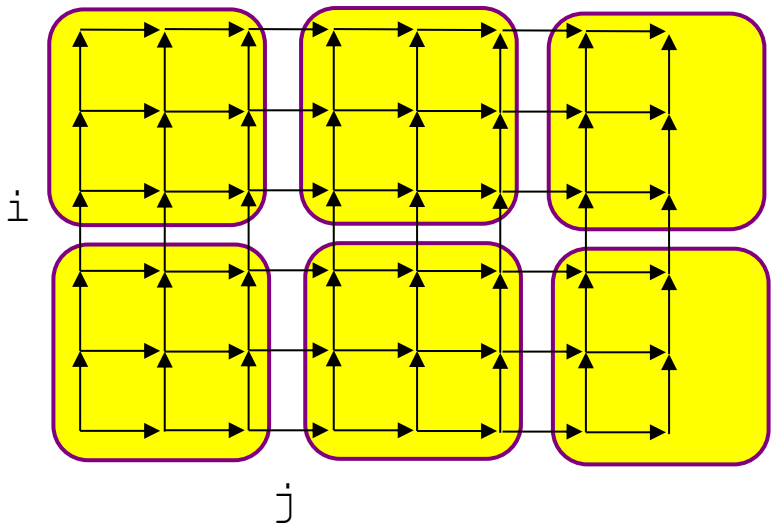
- Apply Affine Transforms to create code with coarsest grain parallelism
- Find the innermost fully permutable loop nest
  - Identify loop with parallelism with consecutive data access (last dimension of the array in C)
  - Permute it innermost

3. Locality

- A Data Locality Optimizing Algorithm
  M. E. Wolf and M. S. Lam
  In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.
  ACM Most Influential PLDI Paper Award, 2001.
- The Cache Performance and Optimizations of Blocked Algorithms
  M. S. Lam, E. E. Rothberg and M. E. Wolf
  In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
  ACM SIGARCH/SIGPLAN/SIGOPS ASPLOS Influential Paper Award, 2021.
- Design and Evaluation of a Compiler Algorithm for Prefetching (PDF)
  T. C. Mowry, M. S. Lam and A. Gupta
  In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1992.
  ACM SIGARCH/SIGPLAN/SIGOPS ASPLOS Influential Paper Award, 2022.

Blocking for Locality: Intro

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



2D blocking:
- 2D iteration space
  - Each unit is a 2D block

It is not affine

Reduces synchronization

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

Example 1: SOR

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

- Original program

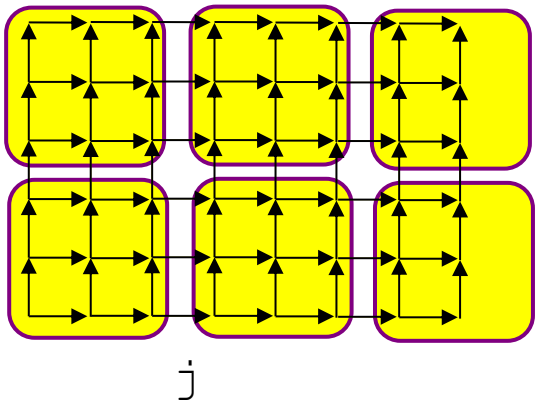
```
for (i = 1; i < 101; i++) {
  for (j = 1; j < 101; j++) {
    A[i,j] = c * (A[i-1,j] + A[i,j-1]) }}}
```

- 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]) }}}
}}}
```

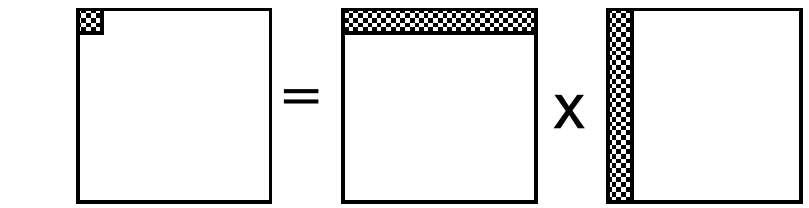


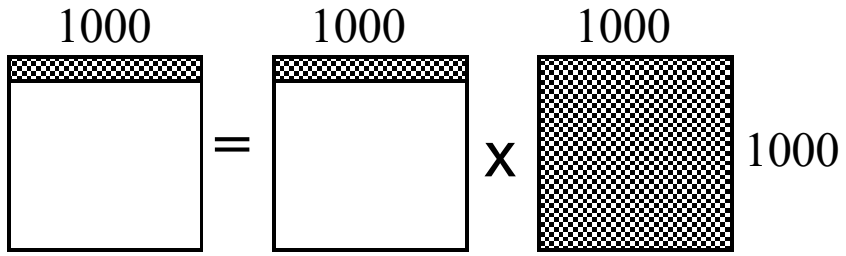
When Do We Block?

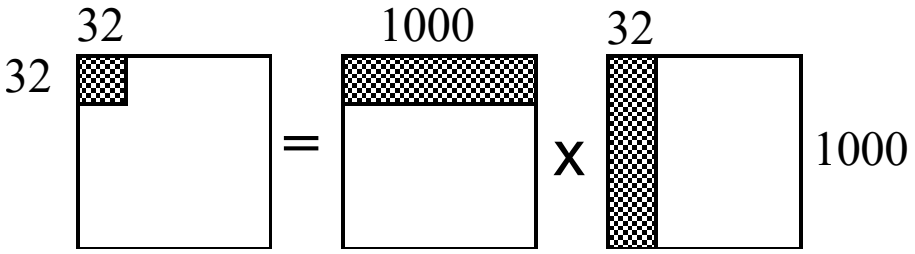
```
for (i = 1; i < 101; i++) {
  for (j = 1; j < 101; j++) {
    A[i,j] = c * (A[i-1,j] + A[i,j-1]) }}}
```

- Kinds of locality:
  - Self reuse: if a reference uses the same location
    - None in this example
  - Self spatial reuse: if a reference uses the same cache line
    - E.g. consecutive in the lowest dimension
    - **`A[i,j], A[i-1,j], A[i,j-1]`** along the j dimension
  - Group reuse: if two references use the same location
    - **`A[i,j], A[i-1,j]`** along the i dimension
    - **`A[i,j], A[i,j-1]`** along the j dimension
    - **`A[i-1,j], A[i,j-1]`** along a diagonal
  - Group spatial reuse: if two references use the same cache line
    - All pairs share group spatial reuse
- SOR: reuse in both i and j dimensions → Block both loops

Example 2: Matrix Multiplication







Data Accessed

1002000

65024

Blocking with Matrix Multiplication

- Original program

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

- Quiz: How many degrees of communication-free parallelism?
- Quiz: How many fully permutable loop nest?

- Analyzing locality
  - **`Z[i,j]:`** self-reuse in dim __, spatial reuse dim __
  - **`X[i,k]:`** self-reuse in dim __, spatial reuse dim __
  - **`Y[k,j]:`** self-reuse in dim __, spatial reuse dim __
  - **`Z[i,j], Z[i,j]:`** group reuse in dim __, spatial reuse in dim __

- Quiz: What loops should we block?

Locality Optimization

- Determine the dimensions of reuse
  - Compute the null space of the array access
  - E.g. self reuse of reference Fi+f:
    - Given two iterations refer to the same location, i, i'

      Fi + f = Fi' + f

      F(i – I') = 0

      Solution is the null space of F
    - Example: `Z[i,j]` self-reuse in the dim K

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

  - The null space determines the dims where locality is found
  - (You don't have to worry about the details of this part, as long as you understand how to hand optimize simple codes)

- Block all fully permutable loop dimensions with reuse

Blocking with Matrix Multiplication

- Original program

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

- Stripmine 2 outer loops

```
for (ii = 0; ii < 100; ii = ii+10) {
  for (i = ii; i < ii+10; i++) {
    for (jj = 0; jj < 100; jj = jj+10) {
      for (j = jj; j < jj+10; j++) {
         for (k = 0; k < 100; k++) {
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
    }}}
```

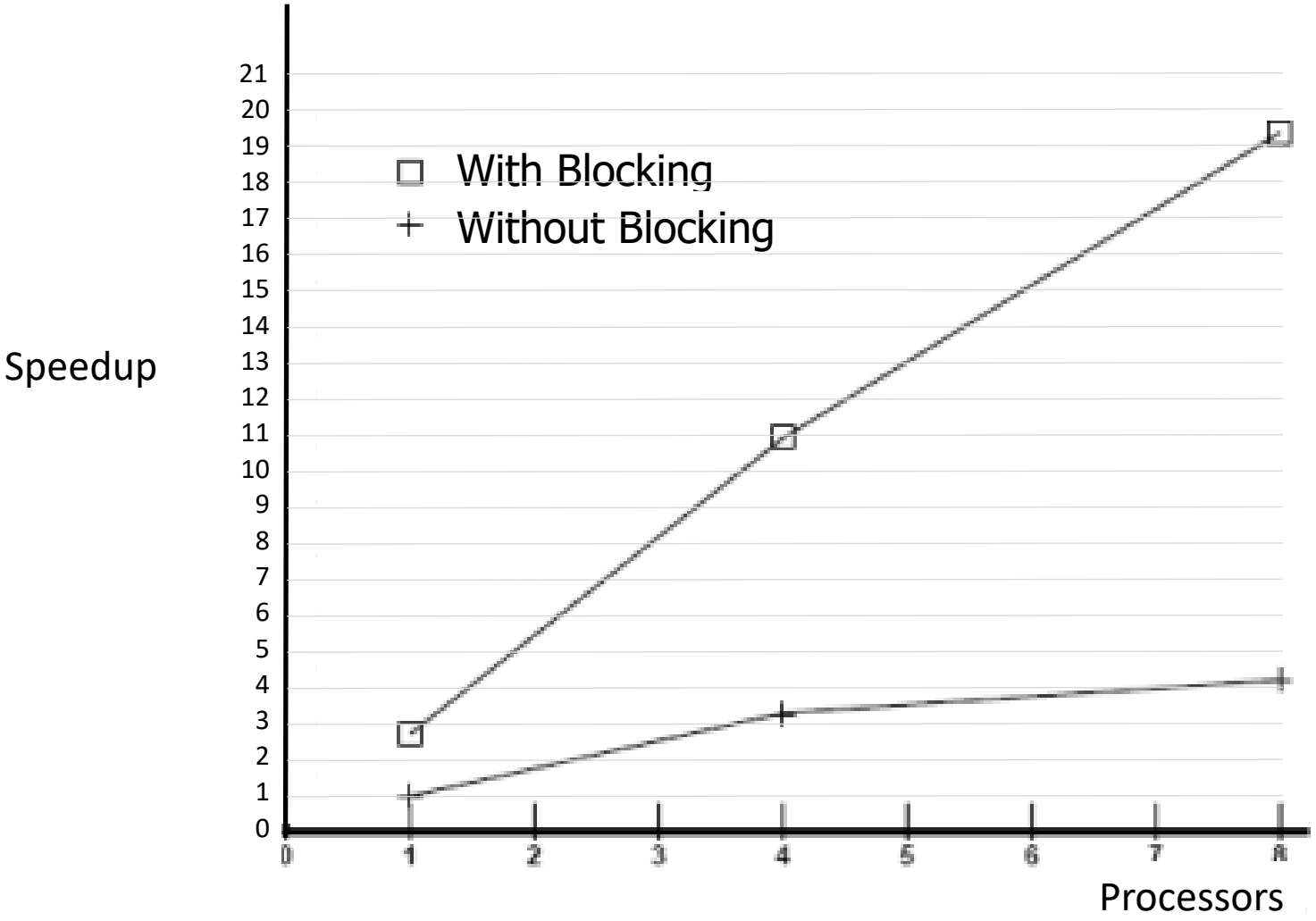
- Permute loops

```
for (ii = 0; ii < 100; ii = ii+10) {
 for (jj = 0; jj < 100; jj = jj+10) {
   for (k = 0; k < 100; k++) {
     for (i = ii; i < ii+10); i++) {
      for (j = jj; j < jj+10; j++) {
        Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
}}}
```

1. Since the i and j loops are blocked, reuse in k loop has locality

2. Innermost loop has instruction level parallelism and supports SIMD operations

Experimental Results



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

Outer loop parallelism

Fusion of multi-dimensional permutable loops for improved locality

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]); }
```

Irregular loop nest
$O(N^3)$ computation
j loop and m loop are do all loops

Too many barriers and data communication

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]); }
```

All dependences
point forward in all dims
→ Full permutable loop nest

Solving the inequalities
finds the mapping from indices to new I,j,k fully permutable loop nests

Iteration **i**
  Writes into row **i**
E.g. **i** = 5:
  Writes [5,1], [5,2], …[5,5]
  For each [5, **j**],
    Reads **1:j** columns
      from row 5 and row **j**
    the last element [5,5]
      uses a sqrt, not div
E.g. [5,4]

```
X[5,4] = (X[5,4]-
X[5,1]*X[4,1]-
X[5,2]*X[4,2]-
X[5,3]*X[4,3]) / X[4,4];

X[5,5] = sqrt (X[5,5]-
X[5,1]*X[5,1]-
X[5,2]*X[5,2]-
X[5,3]*X[5,3]-
X[5,4]*X[5,4]) ;
```

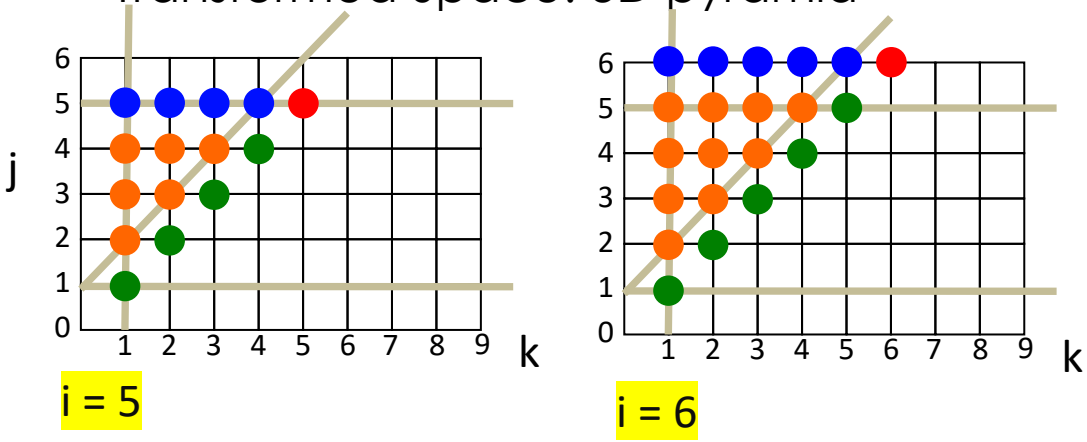
Example: Cholesky Decomposition

```
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]); }
```

2D parallelism: p1= i; p2 = j

```
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]); }}}
```

Transformed Space: 3D pyramid

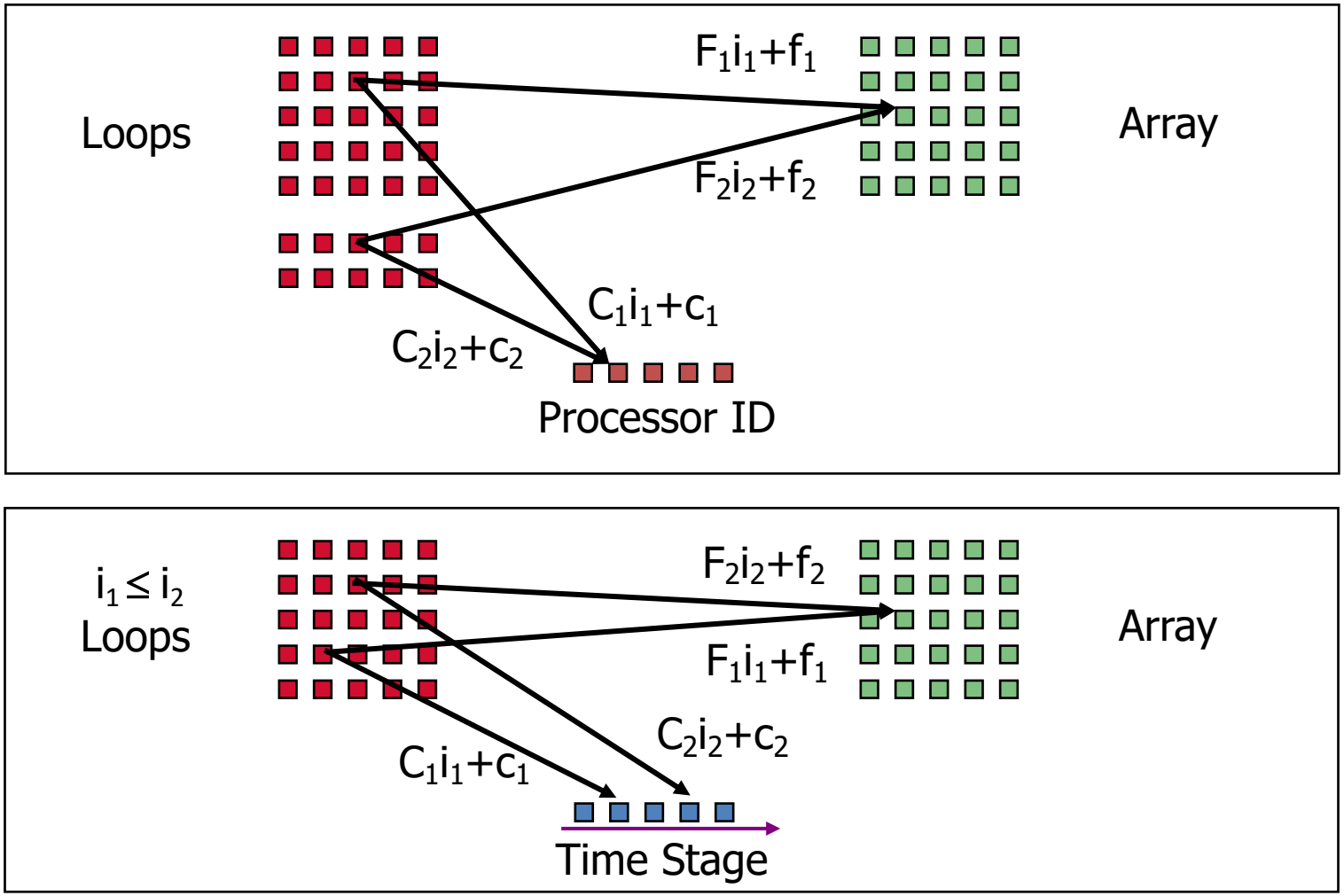


Many non-perfectly nested loops

3-deep fully permutable loop nest with 2-dimensional parallelism

Blocking → high performance

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: Can customize for different machine models