

Data Dependence, Parallelization, and Locality Enhancement

(courtesy of Tarek Abdelrahman, University of Toronto)

Data Dependence

$$\begin{array}{l} S_1: A = 1.0 \\ S_2: B = A + 2.0 \\ S_3: A = C - D \\ \vdots \\ S_4: A = B/C \end{array}$$

We define four types of data dependence.

- **Flow (true) dependence:** a statement S_i precedes a statement S_j in execution and S_i computes a data value that S_j uses.
- Implies that S_i must execute before S_j .

$$S_i \delta^f S_j \quad (S_i \delta^f S_2 \text{ and } S_2 \delta^f S_4)$$

Data Dependence

$$\begin{array}{l} S_1: A = 1.0 \\ S_2: B = A + 2.0 \\ S_3: A = C - D \\ \vdots \\ S_4: A = B/C \end{array}$$

We define four types of data dependence.

- **Anti dependence:** a statement S_i precedes a statement S_j in execution and S_i uses a data value that S_j computes.
- It implies that S_i must be executed before S_j .

$$S_i \delta^a S_j \quad (S_2 \delta^a S_3)$$

Data Dependence

$$\begin{array}{l} S_1: A = 1.0 \\ S_2: B = A + 2.0 \\ S_3: A = C - D \\ \vdots \\ S_4: A = B/C \end{array}$$

We define four types of data dependence.

- **Output dependence:** a statement S_i precedes a statement S_j in execution and S_i computes a data value that S_j also computes.
- It implies that S_i must be executed before S_j .

$$S_i \delta^o S_j \quad (S_1 \delta^o S_3 \text{ and } S_3 \delta^o S_4)$$

Data Dependence

S_1 : $A = 1.0$
 S_2 : $B = A + 2.0$
 S_3 : $A = C - D$
 \vdots
 S_4 : $A = B/C$

We define four types of data dependence.

- **Input dependence**: a statement S_i precedes a statement S_j in execution and S_i uses a data value that S_j also uses.
- Does this imply that S_i must execute before S_j ?

$S_i \delta^I S_j$ ($S_3 \delta^I S_4$)

Data Dependence (continued)

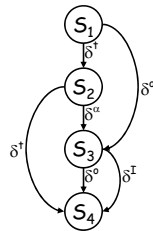
- The dependence is said to **flow** from S_i to S_j because S_i precedes S_j in execution.
- S_i is said to be the **source** of the dependence. S_j is said to be the **sink** of the dependence.
- The only "true" dependence is flow dependence; it represents the flow of data in the program.
- The other types of dependence are caused by programming style; they may be eliminated by re-naming.

S_1 : $A = 1.0$
 S_2 : $B = A + 2.0$
 S_3 : $A1 = C - D$
 \vdots
 S_4 : $A2 = B/C$

Data Dependence (continued)

- Data dependence in a program may be represented using a **dependence graph** $G=(V,E)$, where the nodes V represent statements in the program and the directed edges E represent dependence relations.

S_1 : $A = 1.0$
 S_2 : $B = A + 2.0$
 S_3 : $A = C - D$
 \vdots
 S_4 : $A = B/C$



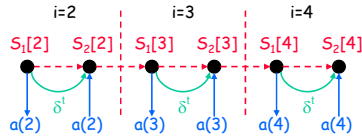
Value or Location?

- There are two ways a dependence is defined: **value-oriented** or **location-oriented**.

S_1 : $A = 1.0$
 S_2 : $B = A + 2.0$
 S_3 : $A = C - D$
 \vdots
 S_4 : $A = B/C$

Example 1

```
do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i)
end do
```

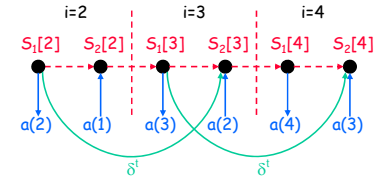


- There is an instance of S_1 that precedes an instance of S_2 in execution and S_1 produces data that S_2 consumes.
- S_1 is the **source** of the dependence; S_2 is the **sink** of the dependence.
- The dependence flows between instances of statements in the same iteration (**loop-independent** dependence).
- The number of iterations between source and sink (**dependence distance**) is 0. The **dependence direction** is $=$.

$$S_1 \delta_0^= S_2 \quad \text{or} \quad S_1 \delta_0^= S_2$$

Example 2

```
do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i-1)
end do
```

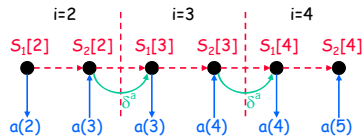


- There is an instance of S_1 that precedes an instance of S_2 in execution and S_1 produces data that S_2 consumes.
- S_1 is the source of the dependence; S_2 is the sink of the dependence.
- The dependence flows between instances of statements in different iterations (**loop-carried** dependence).
- The dependence distance is 1. The direction is positive ($<$).

$$S_1 \delta_1^< S_2 \quad \text{or} \quad S_1 \delta_1^< S_2$$

Example 3

```
do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i+1)
end do
```



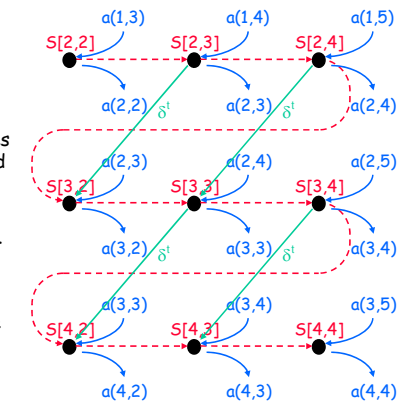
- There is an instance of S_2 that precedes an instance of S_1 in execution and S_2 consumes data that S_1 produces.
- S_2 is the source of the dependence; S_1 is the sink of the dependence.
- The dependence is loop-carried.
- The dependence distance is 1.

$$S_2 \delta_1^> S_1 \quad \text{or} \quad S_2 \delta_1^> S_1$$

- Are you sure you know why it is $S_2 \delta_1^> S_1$ even though S_1 appears before S_2 in the code?

Example 4

```
do i = 2, 4
do j = 2, 4
S: a(i,j) = a(i-1,j+1)
end do
end do
```



- An instance of S precedes another instance of S and S produces data that S consumes.
- S is both source and sink.
- The dependence is loop-carried.
- The dependence distance is $(1,-1)$.

$$S \delta_{(1,-1)}^> S \quad \text{or} \quad S \delta_{(1,-1)}^> S$$

Problem Formulation

- Consider the following **perfect** nest of depth d :

```

do I1 = L1, U1
do I2 = L2, U2
...
do Id = Ld, Ud
a(f1(Ī), f2(Ī), ..., fm(Ī)) = ...
... = a(g1(Ī), g2(Ī), ..., gn(Ī))
enddo
enddo
enddo

Ī = (I1, I2, ..., Id)
L̄ = (L1, L2, ..., Ld)
Ū = (U1, U2, ..., Ud)
L̄ ≤ Ū
    
```

linear functions
 $b_0 + b_1 I_1 + b_2 I_2 + \dots + b_d I_d$

array reference
 $a(\dots, f_k(\vec{I}), \dots)$

subscript position subscript function or subscript expression

Problem Formulation

- Dependence will exist if there exists two iteration vectors \vec{k} and \vec{j} such that $\vec{L} \leq \vec{k} \leq \vec{j} \leq \vec{U}$ and:

$$\begin{aligned}
 & \text{and } f_1(\vec{k}) = g_1(\vec{j}) \\
 & \text{and } f_2(\vec{k}) = g_2(\vec{j}) \\
 & \quad \vdots \\
 & \text{and } f_m(\vec{k}) = g_m(\vec{j})
 \end{aligned}$$

- That is:

$$\begin{aligned}
 & f_1(\vec{k}) - g_1(\vec{j}) = 0 \\
 & \text{and } f_2(\vec{k}) - g_2(\vec{j}) = 0 \\
 & \quad \vdots \\
 & \text{and } f_m(\vec{k}) - g_m(\vec{j}) = 0
 \end{aligned}$$

Problem Formulation - Example

```

do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i-1)
end do
    
```

- Does there exist two iteration vectors i_1 and i_2 , such that $2 \leq i_1 \leq i_2 \leq 4$ and such that:

$$i_1 = i_2 - 1?$$

- Answer: yes; $i_1=2$ & $i_2=3$ and $i_1=3$ & $i_2=4$.
- Hence, there is dependence!
- The dependence distance vector is $i_2 - i_1 = 1$.
- The dependence direction vector is $\text{sign}(1) = <$.

Problem Formulation - Example

```

do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i+1)
end do
    
```

- Does there exist two iteration vectors i_1 and i_2 , such that $2 \leq i_1 \leq i_2 \leq 4$ and such that:

$$i_1 = i_2 + 1?$$

- Answer: yes; $i_1=3$ & $i_2=2$ and $i_1=4$ & $i_2=3$. (But, but!).
- Hence, there is dependence!
- The dependence distance vector is $i_2 - i_1 = -1$.
- The dependence direction vector is $\text{sign}(-1) = >$.
- Is this possible?

Problem Formulation - Example

```
do i = 1, 10
  S1: a(2*i) = b(i) + c(i)
  S2: d(i) = a(2*i+1)
end do
```

- Does there exist two iteration vectors i_1 and i_2 , such that $1 \leq i_1 \leq i_2 \leq 10$ and such that:

$$2 * i_1 = 2 * i_2 + 1?$$

- Answer: no; $2 * i_1$ is even & $2 * i_2 + 1$ is odd.
- Hence, there is no dependence!

Problem Formulation

- Dependence testing is equivalent to an **integer linear programming (ILP)** problem of $2d$ variables & $m+d$ constraint!
- An algorithm that determines if there exists two iteration vectors \vec{k} and \vec{j} that satisfies these constraints is called a **dependence tester**.
- The dependence distance vector is given by $\vec{j} - \vec{k}$.
- The dependence direction vector is given by $\text{sign}(\vec{j} - \vec{k})$.
- Dependence testing is NP-complete!
- A dependence test that reports dependence only when there is dependence is said to be **exact**. Otherwise it is **in-exact**.
- A dependence test must be **conservative**; if the existence of dependence cannot be ascertained, dependence must be assumed.

Dependence Testers

- Lamport's Test.
- GCD Test.
- Banerjee's Inequalities.
- Generalized GCD Test.
- Power Test.
- I-Test.
- Omega Test.
- Delta Test.
- Stanford Test.
- etc...

Lamport's Test

- Lamport's Test is used when there is a single index variable in the subscript expressions, and when the coefficients of the index variable in both expressions are the same.

$$A(\dots, b * i + c_1, \dots) = \dots \\ \dots = A(\dots, b * i + c_2, \dots)$$

- The dependence problem: does there exist i_1 and i_2 , such that $L_i \leq i_1 \leq i_2 \leq U_i$ and such that

$$b * i_1 + c_1 = b * i_2 + c_2? \quad \text{or} \quad i_2 - i_1 = \frac{c_1 - c_2}{b}?$$

- There is integer solution if and only if $\frac{c_1 - c_2}{b}$ is integer.
- The dependence distance is $d = \frac{c_1 - c_2}{b}$ if $L_i \leq |d| \leq U_i$.
- $d > 0 \Rightarrow$ true dependence.
- $d = 0 \Rightarrow$ loop independent dependence.
- $d < 0 \Rightarrow$ anti dependence.

Lampport's Test - Example

```

do i = 1, n
  do j = 1, n
    S: a(i,j) = a(i-1,j+1)
  end do
end do

```

- $i_1 = i_2 - 1?$

$b = 1; c_1 = 0; c_2 = -1$

$$\frac{c_1 - c_2}{b} = 1$$

There is dependence.
Distance (i) is 1.

- $j_1 = j_2 + 1?$

$b = 1; c_1 = 0; c_2 = 1$

$$\frac{c_1 - c_2}{b} = -1$$

There is dependence.
Distance (j) is -1.

S $\vec{\delta}_{(1,-1)}$ S or S $\vec{\delta}_{(<, >)}$ S

Carnegie Mellon
Todd C. Mowry

Lampport's Test - Example

```

do i = 1, n
  do j = 1, n
    S: a(i,2*j) = a(i-1,2*j+1)
  end do
end do

```

- $i_1 = i_2 - 1?$

$b = 1; c_1 = 0; c_2 = -1$

$$\frac{c_1 - c_2}{b} = 1$$

There is dependence.
Distance (i) is 1.

- $2*j_1 = 2*j_2 + 1?$

$b = 2; c_1 = 0; c_2 = 1$

$$\frac{c_1 - c_2}{b} = -\frac{1}{2}$$

There is no dependence.

?

There is no dependence!

Carnegie Mellon
Todd C. Mowry

GCD Test

- Given the following equation:

$$\sum_{i=1}^n a_i x_i = c \quad a_i \text{'s and } c \text{ are integers}$$

an integer solution exists if and only if:

$$\gcd(a_1, a_2, \dots, a_n) \text{ divides } c$$

- Problems:
 - ignores loop bounds.
 - gives no information on distance or direction of dependence.
 - often gcd(.....) is 1 which always divides c, resulting in false dependences.

Carnegie Mellon
Todd C. Mowry

GCD Test - Example

```

do i = 1, 10
  S1: a(2*i) = b(i) + c(i)
  S2: d(i) = a(2*i-1)
end do

```

- Does there exist two iteration vectors i_1 and i_2 , such that $1 \leq i_1 \leq i_2 \leq 10$ and such that:

$$2*i_1 = 2*i_2 - 1?$$
 or

$$2*i_2 - 2*i_1 = 1?$$
- There will be an integer solution if and only if **gcd(2,-2) divides 1**.
- This is not the case, and hence, there is no dependence!

Carnegie Mellon
Todd C. Mowry

GCD Test Example

```
do i = 1, 10
  S1: a(i) = b(i) + c(i)
  S2: d(i) = a(i-100)
end do
```

- Does there exist two iteration vectors i_1 and i_2 , such that $1 \leq i_1 \leq i_2 \leq 10$ and such that:

$i_1 = i_2 - 100?$
or
 $i_2 - i_1 = 100?$

- There will be an integer solution if and only if **gcd(1,-1) divides 100**.
- This is the case, and hence, there is dependence! Or is there?

Dependence Testing Complications

- Unknown loop bounds.

```
do i = 1, N
  S1: a(i) = a(i+10)
end do
```

What is the relationship between N and 10?

- Triangular loops.

```
do i = 1, N
  do j = 1, i-1
    S: a(i,j) = a(j,i)
  end do
end do
```

Must impose $j < i$ as an additional constraint.

More Complications

- User variables.

```
do i = 1, 10
  S1: a(i) = a(i+k)
end do
```

Same problem as unknown loop bounds, but occur due to some loop transformations (e.g., normalization).

```
do i = L, H
  S1: a(i) = a(i-1)
end do
```



```
do i = 1, H-L
  S1: a(i+L) = a(i+L-1)
end do
```

More Complications

- Scalars.

```
do i = 1, N
  S1: x = a(i)
  S2: b(i) = x
end do
```



```
do i = 1, N
  S1: x(i) = a(i)
  S2: b(i) = x(i)
end do
```

```
j = N-1
do i = 1, N
  S1: a(i) = a(j)
  S2: j = j - 1
end do
```



```
do i = 1, N
  S1: a(i) = a(N-i)
end do
```

```
sum = 0
do i = 1, N
  S1: sum = sum + a(i)
end do
```



```
do i = 1, N
  S1: sum(i) = a(i)
end do
sum += sum(i) i = 1, N
```

Serious Complications

- Aliases.

- Equivalence Statements in Fortran:

```
real a(10,10), b(10)
```

makes b the same as the first column of a.

- Common blocks: Fortran's way of having shared/global variables.

```
common /shared/a,b,c  
:  
:
```

```
subroutine foo (...)  
common /shared/a,b,c
```

```
common /shared/x,y,z
```

Loop Parallelization

- A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
  do j = 2, m-1  
    a(i, j) = ...  
    ... = a(i, j)  
  
    b(i, j) = ...  
    ... = b(i, j-1)  
  
    c(i, j) = ...  
    ... = c(i-1, j)  
  end do  
end do
```

Loop Parallelization

- A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
  do j = 2, m-1  
    a(i, j) = ...  
    ... = a(i, j)  
  
    b(i, j) = ...  
    ... = b(i, j-1)  
  
    c(i, j) = ...  
    ... = c(i-1, j)  
  end do  
end do
```

Loop Parallelization

- A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
  do j = 2, m-1  
    a(i, j) = ...  
    ... = a(i, j)  
  
    b(i, j) = ...  
    ... = b(i, j-1)  
  
    c(i, j) = ...  
    ... = c(i-1, j)  
  end do  
end do
```


Loop Parallelization

- A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```

do i = 2, n-1
  do j = 2, m-1
    a(i, j) = ...
    ... = a(i, j)

    b(i, j) = ...
    ... = b(i, j-1)

    c(i, j) = ...
    ... = c(i-1, j)
  end do
end do

```

$\delta_{<=}^+$

Loop Parallelization

- A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```

do i = 2, n-1
  do j = 2, m-1
    a(i, j) = ...
    ... = a(i, j)

    b(i, j) = ...
    ... = b(i, j-1)

    c(i, j) = ...
    ... = c(i-1, j)
  end do
end do

```

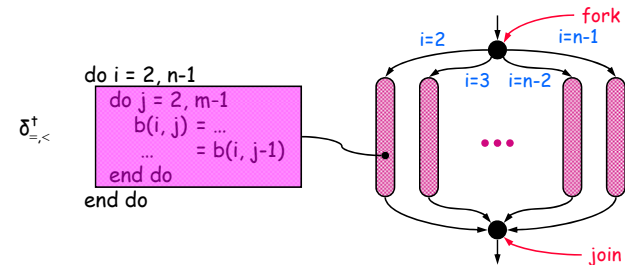
$\delta_{=, <}^+$

- Outermost loop with a non "=" direction carries dependence!

Loop Parallelization

The iterations of a loop may be executed in parallel with one another if and only if no dependences are carried by the loop!

Loop Parallelization - Example



- Iterations of loop j must be executed sequentially, but the iterations of loop i may be executed in parallel.
- Outer loop parallelism.

Loop Parallelization - Example

```

do i = 2, n-1
  do j = 2, m-1
    b(i, j) = ...
    ... = b(i-1, j)
  end do
end do

```

- Iterations of loop i must be executed sequentially, but the iterations of loop j may be executed in parallel.
- Inner loop parallelism.

Carnegie Mellon
Todd C. Mowry

Loop Parallelization - Example

```

do i = 2, n-1
  do j = 2, m-1
    b(i, j) = ...
    ... = b(i-1, j-1)
  end do
end do

```

- Iterations of loop i must be executed sequentially, but the iterations of loop j may be executed in parallel. **Why?**
- Inner loop parallelism.

Carnegie Mellon
Todd C. Mowry

Loop Interchange

Loop interchange changes the order of the loops to improve the spatial locality of a program.

```

do j = 1, n
  do i = 1, n
    ... a(i, j) ...
  end do
end do

```

Carnegie Mellon
Todd C. Mowry

Loop Interchange

Loop interchange changes the order of the loops to improve the spatial locality of a program.

```

do j = 1, n
  do i = 1, n
    ... a(i, j) ...
  end do
end do

```

```

do i = 1, n
  do j = 1, n
    ... a(i, j) ...
  end do
end do

```

Carnegie Mellon
Todd C. Mowry

Loop Interchange

- Loop interchange can improve the granularity of parallelism!

```
do i = 1, n
  do j = 1, n
    a(i,j) = b(i,j)
    c(i,j) = a(i-1,j)
  end do
end do
```

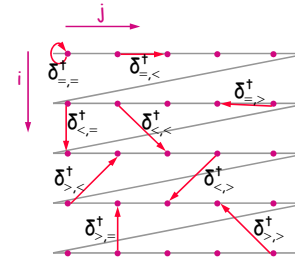
 $\delta_{<=}$

```
do j = 1, n
  do i = 1, n
    a(i,j) = b(i,j)
    c(i,j) = a(i-1,j)
  end do
end do
```

 $\delta_{<}$

Loop Interchange

```
do i = 1, n
  do j = 1, n
    ... a(i,j) ...
  end do
end do
```

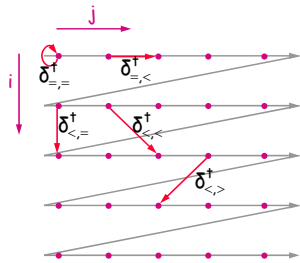


```
do j = 1, n
  do i = 1, n
    ... a(i,j) ...
  end do
end do
```

- When is loop interchange legal?

Loop Interchange

```
do i = 1, n
  do j = 1, n
    ... a(i,j) ...
  end do
end do
```

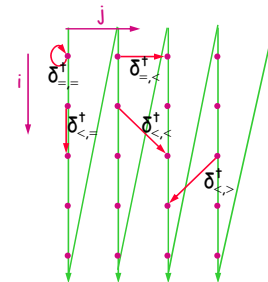


```
do j = 1, n
  do i = 1, n
    ... a(i,j) ...
  end do
end do
```

- When is loop interchange legal?

Loop Interchange

```
do i = 1, n
  do j = 1, n
    ... a(i,j) ...
  end do
end do
```



```
do j = 1, n
  do i = 1, n
    ... a(i,j) ...
  end do
end do
```

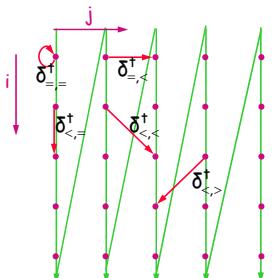
- When is loop interchange legal?

Loop Interchange

```

do i = 1,n
  do j = 1,n
    ... a(i,j) ...
  end do
end do

```



```

do j = 1,n
  do i = 1,n
    ... a(i,j) ...
  end do
end do

```

• When is loop interchange legal? when the "interchanged" dependences remain lexicographically positive!

Carnegie Mellon
Todd C. Mowry

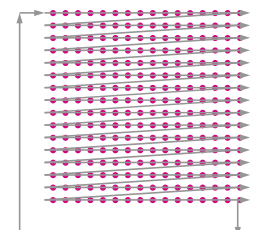
Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```

do t = 1,T
  do i = 1,n
    do j = 1,n
      ... a(i,j) ...
    end do
  end do
end do

```



Carnegie Mellon
Todd C. Mowry

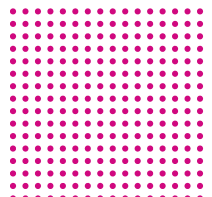
Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```

do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = 1, B
        do j = 1, B
          ... a(ic+i-1, jc+j-1) ...
        end do
      end do
    end do
  end do
end do

```



Carnegie Mellon
Todd C. Mowry

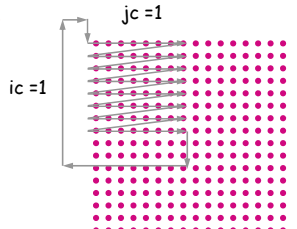
Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```

do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = 1, B
        do j = 1, B
          ... a(ic+i-1, jc+j-1) ...
        end do
      end do
    end do
  end do
end do

```



Carnegie Mellon
Todd C. Mowry

Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```

do ic = 1, n, B
do jc = 1, n, B
do t = 1, T
do i = 1, B
do j = 1, B
... a(ic+i-1, jc+j-1) ...
end do
end do
end do
end do

```

control loops

B: Block size

Carnegie Mellon
Optimizing Compilers: Parallelization -49- Todd C. Mowry

Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```

do ic = 1, n, B
do jc = 1, n, B
do t = 1, T
do i = 1, B
do j = 1, B
... a(ic+i-1, jc+j-1) ...
end do
end do
end do
end do

```

control loops

B: Block size

Carnegie Mellon
Optimizing Compilers: Parallelization -50- Todd C. Mowry

Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```

do ic = 1, n, B
do jc = 1, n, B
do t = 1, T
do i = 1, B
do j = 1, B
... a(ic+i-1, jc+j-1) ...
end do
end do
end do
end do

```

control loops

B: Block size

Carnegie Mellon
Optimizing Compilers: Parallelization -51- Todd C. Mowry

Loop Blocking (Tiling)

```

do t = 1, T
do i = 1, n
do j = 1, n
... a(i, j) ...
end do
end do
end do

```

```

do t = 1, T
do ic = 1, n, B
do i = 1, B
do jc = 1, n, B
do j = 1, B
... a(ic+i-1, jc+j-1) ...
end do
end do
end do
end do

```

```

do ic = 1, n, B
do jc = 1, n, B
do t = 1, T
do i = 1, B
do j = 1, B
... a(ic+i-1, jc+j-1) ...
end do
end do
end do
end do

```

- When is loop blocking legal?

Carnegie Mellon
Optimizing Compilers: Parallelization -52- Todd C. Mowry