Dependence Analysis

(CS 3006)

Department of Computer Science,

National University of Computer & Emerging Sciences,

Islamabad Campus

The Big Picture

1. What are our goals?

- **Simple Goal**: Make execution time as small as possible

2. Which leads to:

– Achieve **execution** of **many** (all, in the best case) **instructions** in **parallel**

– But, you have to find **INDEPENDENT** instructions

Data Dependence

- **Data** must be **produced** and **consumed** in the ***correct order***

- **Simple example of data dependence:**

$$S_1 \quad PI = 3.14$$
$$S_2 \quad R = 5.0$$
$$S_3 \quad AREA = PI * R ** 2$$

- Statement ***S3*** cannot be moved before either ***S1*** or ***S2*** without compromising correct results

Motivation

- **DOALL loops**: **loops whose iterations can execute in parallel**

```
for i = 11, 20
      a[i] = a[i] + 3
```

Examples

```
for i = 11, 20
    a[i] = a[i] + 3
```

Parallel

```
for i = 11, 20
  a[i] = a[i-1] + 3
```

NOT Parallel?

```
for i = 11, 20
  a[i] = a[i-10] + 3
```

Parallel ?

Dependence Analysis

- A **dependence** is a **relationship** between **2 computations** that **places constraints** on their **execution order**
- **Dependence analysis identifies** these **constraints**
- **Constraints** are used to **determine** whether a **particular transformation** can be **applied** without changing the computation's **semantics**
- **2 types of dependences**: **control** and **data** dependences
- ***Both of them must be considered*** when parallelizing programs.

Control Dependence

- There is a **control dependence** between **S1** and **S2,** when **S1** **determines whether** **S2** **will be executed or not**
- **Example:**

```
S1   IF (T .NE. 0.0)
S2        A = A / T
S3   CONTINUE
```

- Executing **S2** before **S1** could cause a **divide by zero** exception (in this example).
- **S2 is conditional** upon the execution of the branch in S1.

Data Dependence

- Two **statements** have a **data dependence** if they **cannot be executed simultaneously** due to **conflicting uses of the same data.**

- **Ensure that data is produced and consumed in the right order:**
    1. do not interchange **loads** and **stores** to the same location
    2. **two stores** take place in the **correct order**

- **Formally:**
    - There is a **data dependence from statement S1 to statement S2 (S2 depends on S1)** if:
        - Both statements **access the same memory location** and at least **one of them stores** onto it, and
        - There is a **feasible run-time execution** path from **S1 to S2**

Load/Store Classification

- Dependences classified in terms of load-store order:
  1. True dependences
     - S_2 depends on S_1 is denoted by $\mathbf{S_1\ \delta\ S_2}$

$$\begin{array}{ll} S_1 & X = \ldots \\ S_2 & \ldots = X \end{array}$$

This is a crucial dependence!

  2. Antidependence
     - S_2 depends on S_1 is denoted by $\mathbf{S_1\ \delta^{-1}\ S_2}$

$$\begin{array}{ll} S_1 & \ldots = X \\ S_2 & X = \ldots \end{array}$$

  3. Output dependence
     - S_2 depends on S_1 is denoted by $\mathbf{S_1\ \delta^{0}\ S_2}$

$$\begin{array}{ll} S_1 & X = \ldots \\ S_2 & X = \ldots \end{array}$$

Data Dependence of Scalar Variables

- **True/Flow dependence**

  a =

  = a

- Output dependence

  a =

  a =

- **Anti-dependence**

  = a

  a =

- Input dependence

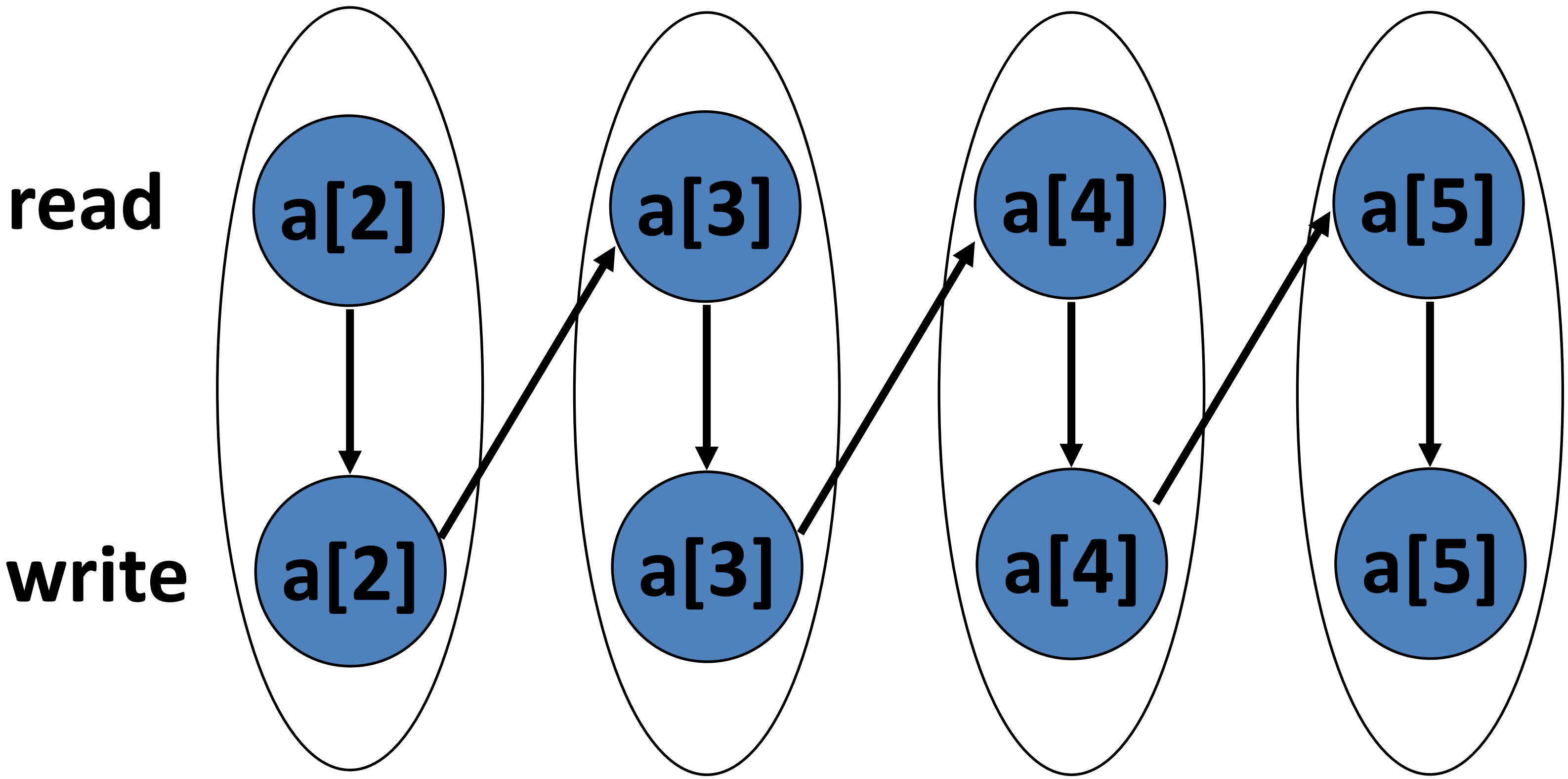
  = a

  = a

- **Only data flow dependences are true dependences.**
- **Anti and output can be removed by renaming**

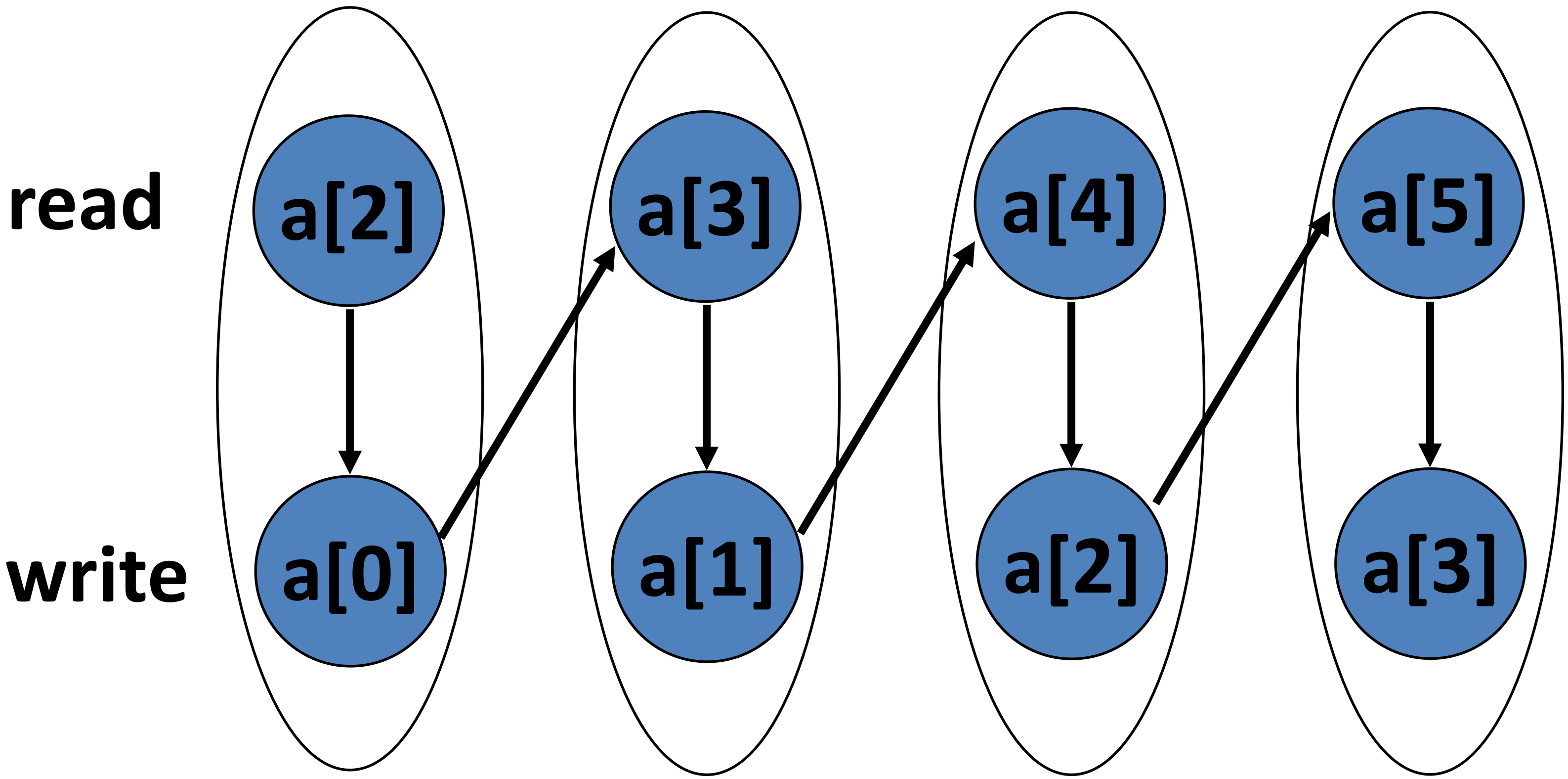
Array Accesses in a Loop

```
for i= 2, 5
    a[i] = a[i] + 3
```



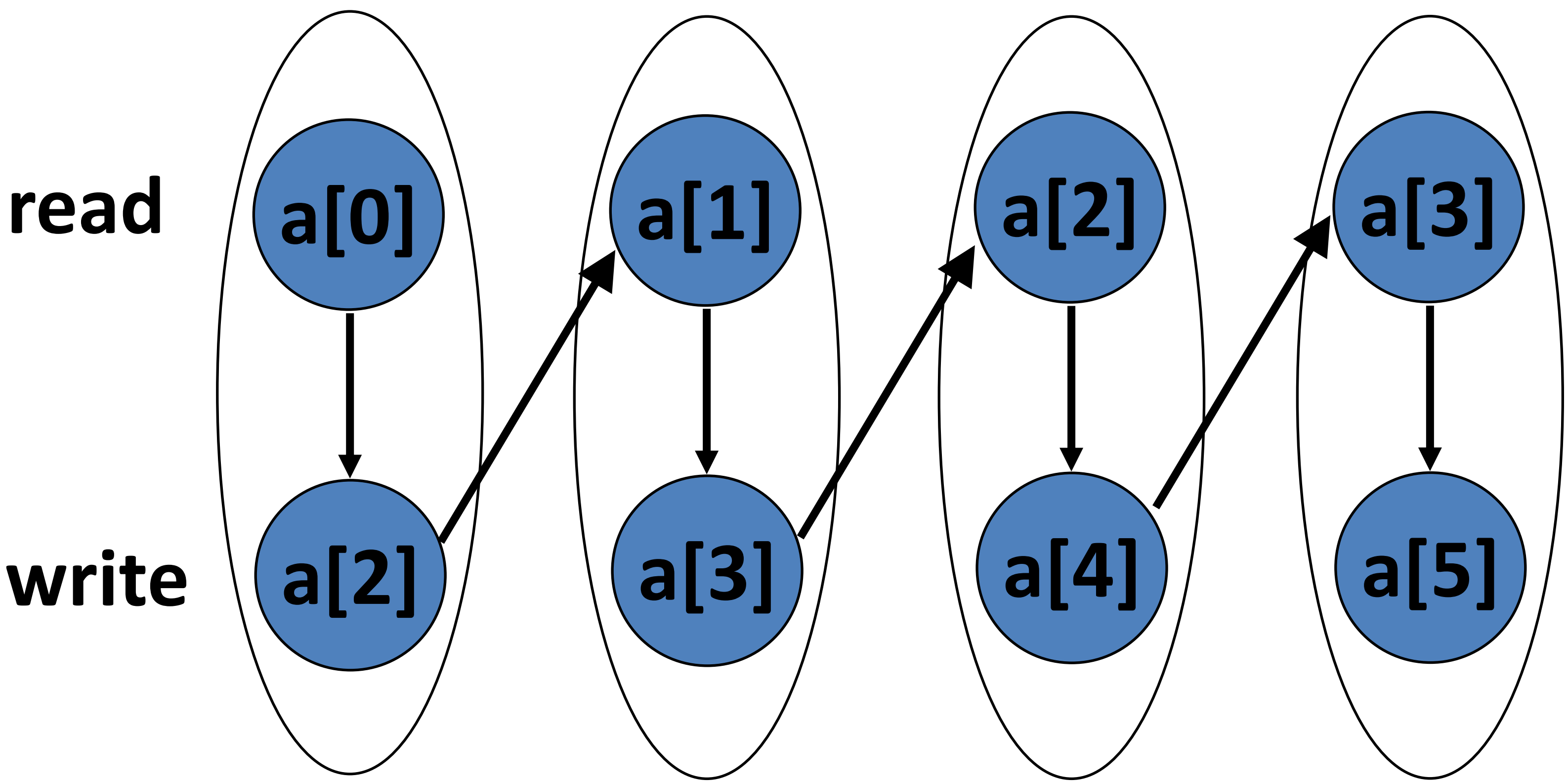
Array Anti-dependence

```
for i= 2, 5
   a[i-2] = a[i] + 3
```



Array True-dependence

```
for i= 2, 5
   a[i] = a[i-2] + 3
```



A Parallel DOALL Loop

- A **loop is fully parallel** if **no dependencies flow across iterations**:

```
DO I = 2, N
   A(I) = A(I) + 1
ENDDO
```

```
DO I = 2, N
  A(I) = A(I-1) + 1
ENDDO
```

- **Parallel loops** are found through **dependence analysis** and **dependence tests**

- Usually **done** at the **source-code level**, **focus is on arrays**

Recognizing DOALL Loops

- **Find** data **dependences** in loop
- **Definition:** a **dependence** is **loop-carried** if it **crosses an iteration boundary**
- **If there are no loop-carried dependences only then loop is parallelizable**

Example: Loop Parallelization

```
Do i=1,n
 A(i)=5*B(i)+A(i)
Enddo
```

```
Do i=1,n
 A(i-1)=5*B(i)+A(i)
Enddo
```

```
Do i=1,n
 tmp=5*B(i)
 A(i)=tmp
Enddo
```

Which of the following loops are parallelizable?

Dependence in Loops

1: A(2) = A(1) + B(1)
2: A(3) = A(2) + B (2)
3: A(4) = A(3) + B(3)
S1 &S1

1: A(3) = A(1) + B(1)
2: A(4) = A(2) + B(2)
3: A(5) = A(3) +B(3)
4: A(6) = A(4) + B(4)

- Let us look at two different loops:

```
      DO I = 1, N
S1      A(I+1) = A(I) + B(I)
      ENDDO
```

```
      DO I = 1, N
S1      A(I+2) = A(I) + B(I)
      ENDDO
```

- In both cases, statement S_1 depends on itself
- However, there is a significant difference.
- We need a formalism to describe and distinguish such dependences

Iteration Numbers

- The **iteration number** of a **loop** is **equal** to the **value** of the **loop index**

- **Definition:**
  - For an arbitrary loop in which the **loop index** ***I*** runs from **L** to **U** in **steps of S**, the iteration number *i* of a specific iteration is equal to the **index value** ***I*** on that **iteration:**

Example:

```
        DO I = 0, 10, 2
S1        <some statement>
        ENDDO
```

Algorithm: Normalizing Loops

Procedure *normalizeLoop*(L_0);
    i = a unique compiler-generated
S1: replace the loop header for L_0
    DO I = L, U, S
    with the adjusted loop header
    DO i = 1, (U – L + S) / S
S2: replace each reference to I within the loop by
    i * S – S + L;
S3: insert a finalization assignment
    I = i * S – S + L;
    immediately after the end of the loop;
end *normalizeLoop;*

Normalizing Loops: Examples

Example: before loop normalization

```
        DO I = 3, 11, 2
S1        A(I) = A(I+1)+ 10
        ENDDO
```

Example: after loop normalization

```
        DO In = 1, 5
S1        A(2*In+1) = A(2*In+2) + 10
        ENDDO
    I=In*2-2+3
```

Normalizing Loops: Examples

normalize so that the loop

- starts at 1 and
- have stride 1

```
L:       do I = 1000, 1, -1
                A(I) = ...
         end do
```

Normalizing Loops: Examples

normalize so that the loop

- starts at 1 and
- have stride 1

```
L:      do I = 1000, 1, -1
                A(I) = ...
        end do
```

normalize

```
Lnorm:  do $I = 1, ((1-1000+(-1)/(-1)
                A(1000 + ($I-1)*(-1)=...
        end do
        I = 1000 + MAX((1-1000+(-1))/(-1)),0)*(-1)
```

simplify

```
Lnorm : do $I = 1, 1000
                A(1001 -$I) = ...
        end do
        I = 0
```

Normalized Iteration Space

```
DO I = 100, 20, -10
  A(I) = B(100-I) + C(I/5)
ENDDO
```

```
DO i = 1, 9
  A(110-10*i) = B(10*i-10)+ C(22-2*i)
ENDDO
```

Iteration Vectors

What do we do for nested loops?

- Need to consider the nesting level of a loop
- Nesting level of a loop is equal to one more than the number of loops that enclose it.
- Given a nest of n loops, the iteration vector *i* of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.
- Thus, the iteration vector is: $\{i_1, i_2, \ldots, i_n\}$ where i_k, $1 \le k \le n$ represents the iteration number for the loop at nesting level k

Nesting level 1

do $I_1 = L_1, U_1$

Nesting level 2

  do $I_2 = L_2, U_2$

    $\ddots$

      do $I_n = L_n, U_n$

        $a(f_1(\vec{I}), f_2(\vec{I}), \cdots, f_m(\vec{I})) = \cdots$

        $\cdots = a(g_1(\vec{I}), g_2(\vec{I}), \cdots, g_m(\vec{I}))$

      enddo

    $\iddots$

  enddo

enddo

Iteration Vectors - Example

Example:

```
DO I = 1, 2
   DO J = 1, 2
S1        <some statement>
   ENDDO
ENDDO
```

- The iteration vector $S_1[(2, 1)]$ denotes the instance of S_1 executed during the 2nd iteration of the I loop and the 1st iteration of the J loop

Iteration Space

Iteration Space: The **set** of all **possible iteration vectors** for a statement.

Example:

```
DO I = 1, 2
   DO J = 1, 2
S1       <some statement>
   ENDDO
ENDDO
```

- The iteration space for S_1 is $\{ (1,1), (1,2), (2,1), (2,2) \}$

Ordering of iteration vectors

- Useful to define an **ordering** for **iteration vectors**

- ***i*** is a vector and $\mathbf{i_k}$ is the **k-th element** of ***i*** and **i[1:k]** is a **k-vector** consisting of the **leftmost k elements** of ***i***

- Define an **order**
  - Iteration ***i*** precedes iteration ***j***, denoted **i < j**, iff:

$$i[1:q] = j[1:q] \text{ and } i[q+1] < j[q+1] \text{ with } 0 <= q < n$$

Formal Definition: Loop Dependence

Theorem: Loop Dependence

- There exists a **dependence** from statements **S1** to statement **S2** in a common **nest of loops** **if and only if** there exist **two iteration vectors** ***i*** and ***j*** for the nest, such that:

    1. ***i < j*** or ***i = j*** and there is a **path** from **S1** to **S2** in the **body of the loop**,
    2. statement **S1 accesses** memory **location *M*** on iteration ***i*** and **statement S2** accesses **location *M*** on iteration ***j***, and
    3. one of these **accesses** is a **write**

Transformations

- We call a **transformation safe** if the **transformed program** has the **same "meaning"** as the **original program**

- But, what is the "**meaning**" of a **program?**

"Meaning" of a program:

- **Two computations** are **equivalent** if, on the **same inputs** they **produce** the **same outputs** in the **same order.**

- Of course the **performance** of the program **may change**!

Re-Ordering Transformations

- A **reordering transformation** is any program **transformation** that **merely changes** the **order of execution** of the **code**, **without adding** or **deleting any executions** of any **statements**.

- A **reordering transformation** does **not eliminate dependences.**

- **However,** it can **change** the **ordering** of **the dependence**:
    - e.g. change from **true** $\rightarrow$ **anti-dependence** or **vice versa**

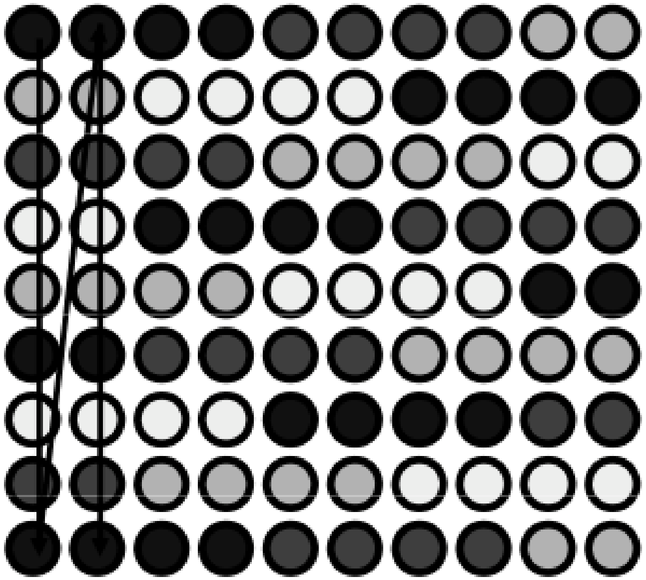
Re-Ordering Transformations

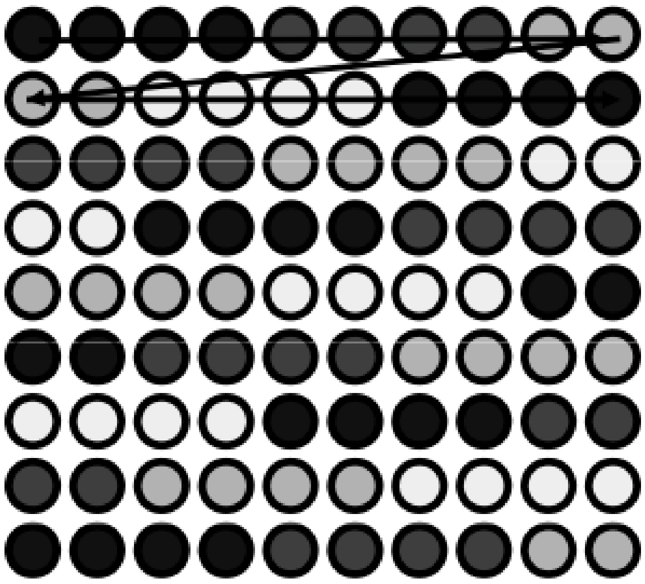
- A **reordering transformation** **preserves a dependence** if it preserves the **relative execution order** of the ***source*** and ***sink*** of that **dependence**.

- **Reordering transformations** changing the **control flow** can be applied to improve the code **unless any dependence is violated.**

Example: Improving Data Locality

```
Do j=1,n
    do i=1,m
      b(i,j)=5.0
    enddo
enddo
```





```
Do i=1,n
    do j=1,m
      b(i,j)=5.0
    enddo
enddo
```

Fundamental Theorem of Dependence

- **Fundamental Theorem of Dependence:**
  - ✧ Any **reordering transformation** that **preserves every dependence** in a **program preserves the meaning of that program**

- A **transformation** is said to be ***valid*** for the program to which it applies **if it preserves all dependences** in the program.

- A **valid transformation preserves** the **order of loads and stores** to every **memory location** in the program – **only input accesses can be reordered.**

Distance Vectors

- **Distance vectors** describe **dependences among iterations**.

- A **dependence** is **loop carried** or **independent** is crucial to determine --> a **loop can be executed** in **parallel or not**?

- They are **important to reason** about **dependences** inside of loops.

- Consider a **dependence** in a loop **nest of *n* loops**

$$S_1 \, \delta \, S_2$$

- ***S_2 depends on S_1***:
  - Statement **S1** on iteration ***i*** is the **source** of the **dependence**
  - Statement **S2** on iteration ***j*** is the **sink** of the **dependence**

Distance Vectors

- The **distance vector d(i, j)** is a **vector** of ***length n*** such that:

$$d(i,j)_k = j_k - i_k$$

- We shall normalize **distance vectors** for loops in which the index **step size is not equal to 1**.

- Note that **distance vectors** describe dependences among **iterations not** among **array elements**.

Direction Vectors

Suppose that there is a **dependence** from **statement S1** on **iteration *i*** of a loop nest of *n* loops and statement **S2 on iteration *j***, then the ***dependence direction vector D*(*i,j*)** is defined as a **vector of length *n*** such that:

$$\boldsymbol{D(i,j)}_k = \begin{cases} \text{“<” if } \boldsymbol{d(i,j)}_k > 0 \\ \text{“=” if } \boldsymbol{d(i,j)}_k = 0 \\ \text{“>” if } \boldsymbol{d(i,j)}_k < 0 \end{cases}$$

$$d(i,j)_k = j_k - i_k$$

Direction Vectors – Example1

```
      DO I = 1, 10
S1        A(2*I) = B(I) + 1
S2        C(I) = A(I)
      ENDDO
```

Iteration Vector	I	S_1	S_2
1	1	A(2)=	=A(1)
2	2	A(4)=	=A(2)
3	3	A(6)=	=A(3)
4	4	A(8)=	=A(4)
5	5	A(10)=	=A(5)
6	6	A(12)=	=A(6)
7	7	A(14)=	=A(7)
8	8	A(16)=	=A(8)
9	9	A(18)=	=A(9)
10	10	A(20)=	=A(10)

Direction Vectors – Example1

```
      DO I = 1, 10
S1        A(2*I) = B(I) + 1
S2        C(I) = A(I)
      ENDDO
```

Iteration Vector	I	S_1	S_2
1	1	A(2)=	=A(1)
2	2	A(4)=	=A(2)
3	3	A(6)=	=A(3)
4	4	A(8)=	=A(4)
5	5	A(10)=	=A(5)
6	6	A(12)=	=A(6)
7	7	A(14)=	=A(7)
8	8	A(16)=	=A(8)
9	9	A(18)=	=A(9)
10	10	A(20)=	=A(10)

dependence relation	array element
$S_1[1]\ \delta\ S_2[2]$	A(2)
$S_1[2]\ \delta\ S_2[4]$	A(4)
$S_1[3]\ \delta\ S_2[6]$	A(6)
$S_1[4]\ \delta\ S_2[8]$	A(8)
$S_1[5]\ \delta\ S_2[10]$	A(10)

- S_2 is true dependent on S_1.
- Distance Vector: (*) as distance varies from 1 to 5
- Direction Vector: (<)

Direction Vectors – Example2

Example:

```
DO I = 1, N
    DO J = 1, M
        DO K = 1, L
S1          A(I+1, J, K-1) = A(I, J, K) + 10
        ENDDO
    ENDDO
ENDDO
```

Direction Vectors – Example2

Example:

```
DO I = 1, N
    DO J = 1, M
```

TWO Sample Iterations:

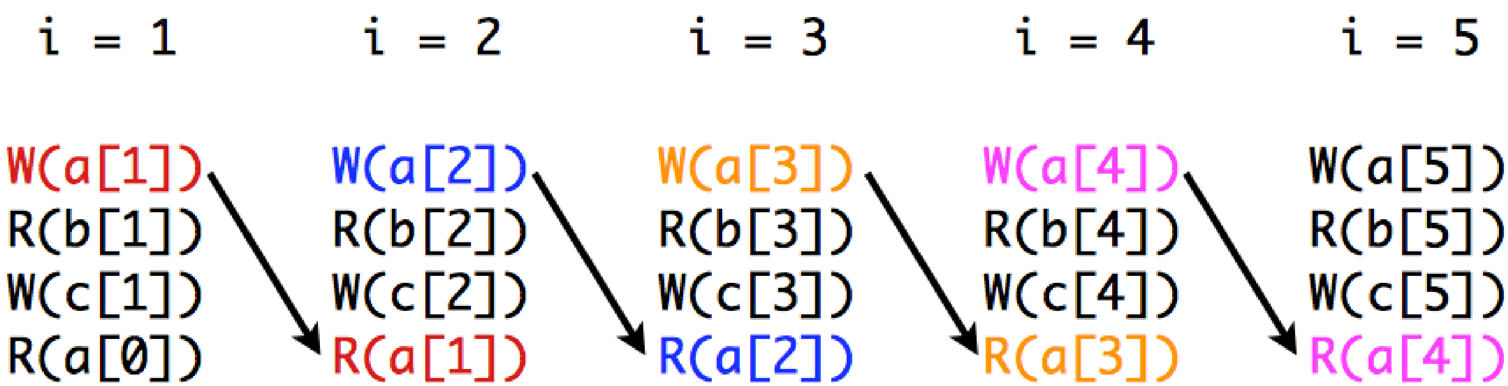
Source Iteration I=2, J=2, K=2 accesses the data item **A(3,2,1)**
Sink iteration I=3, J=2, K=1 accesses the data item **A(3,2,1)**
Sink-Source = (3,2,1) – (2,2,2) = (1,0,-1)

- S_1 has a true dependence on itself.
- Distance Vector: (1, 0, -1)
- Direction Vector: (<, =, >)

Source and Sink of Dependence

- **Dependence source** is the **earlier statement** (the **statement at the tail of the dependence arrow**)

- **Dependence sink** is the **later statement** (the **statement at the head of the dependence arrow**)



- **Dependences can only go forward in time**: **always from an earlier iteration to a later iteration.**

Direction Vectors

- A **dependence cannot exist** if it has a **direction vector** whose **leftmost non "="** component **is not "<"**
  - As this would imply that the **sink of the dependence occurs before the source (<u>which is not true</u>)**

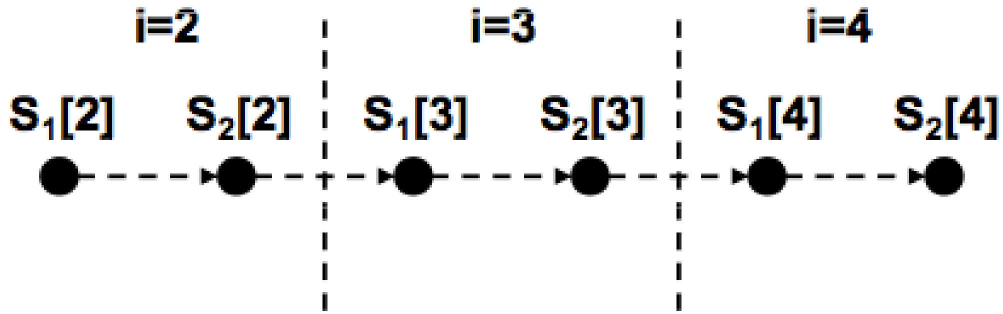
- **How many dependences are there between a pair of statements in a loop nest?**
  - ✧ **One dependence** for **every statement instance** that is the **source** of a **dependence** to **another statement instance** in the **same loop nest**.
  - ✧ **Compiler collect** only the **different direction vectors** for **every different pair of statements**.

Example-1

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

Example-1

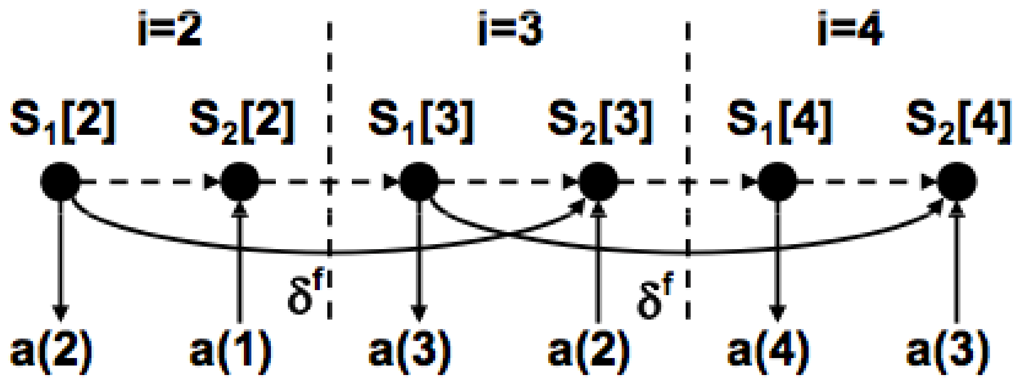
```
     do i = 2, 4
S1:    a(i) = b(i) + c(i)
S2:    d(i) = a(i)
    end do
```



- There is an **instance** of ***S1*** that **precedes an instance** of ***S2*** in execution, and **S1 produces data** that **S2 consumes**.
- **S1 is the source** of the dependence; **S2 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**. Dependence **direction is (=)**.

Example-2

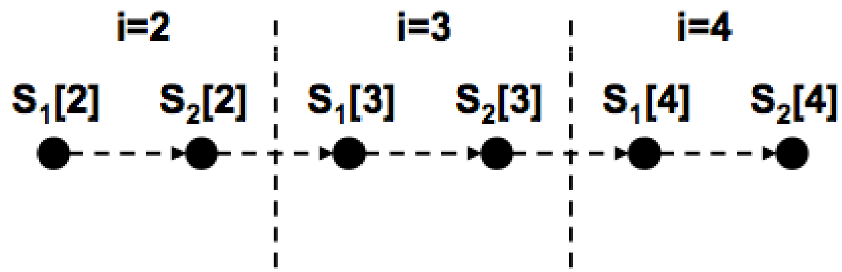
do i = 2, 4
S_1: a(i) = b(i) + c(i)
S_2: d(i) = a(i-1)
end do



- There is an instance of ***S1*** that precedes an instance of ***S2*** in execution, and S1 produces data that S2 consumes.
- **S1 is the source** of the dependence; **S2 is the sink** of the dependence.
- The dependence flows between iterations (***loop-carried dependence***).
- The number of ***iterations*** between **source** and **sink** ***(dependence distance)*** is **1**. Dependence **direction is (<)**.

Example-3

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

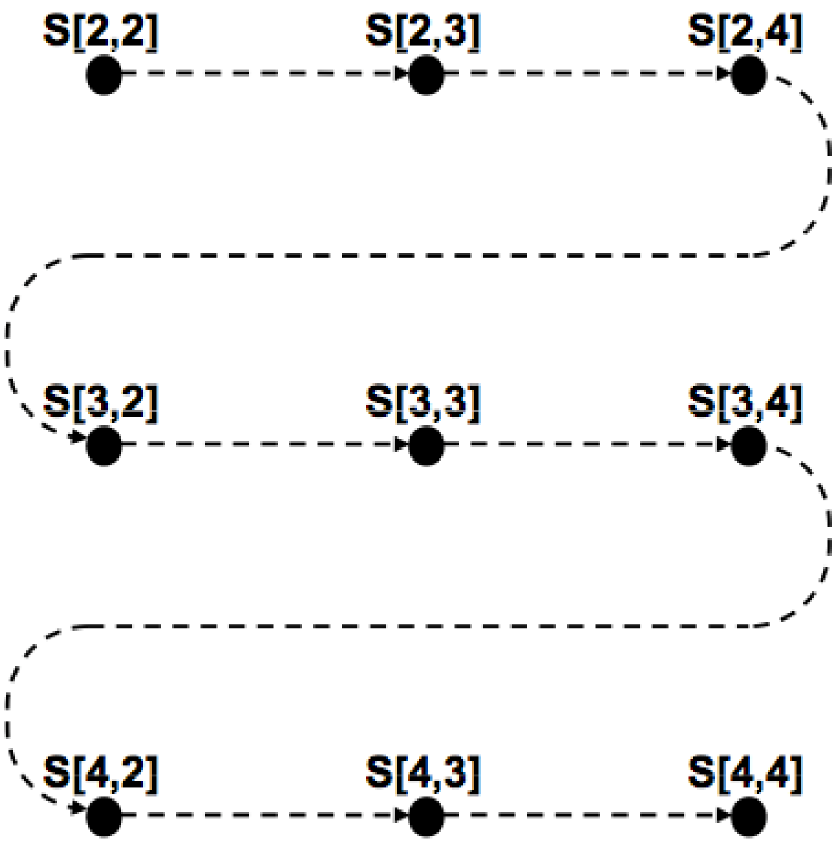


- There is an instance of S_2 that precedes an instance of S_1 in execution anc S_2 consumes data that S_1 reassigns.
- S_2 is the source of the dependence with iteration vector I; S_1 is the sink with iteration vector J
- The dependence is loop-carried.
- The distance is 1. The direction is positive with a direction vector (<).
- S_1 is before S_2 in the loop body, so why < direction? $\mathbf{S_2\ \delta^{-1}\ S_1}$

Iteration Vector	I	S_1	S_2
2	2	A(2)=	=A(3)
3	3	A(3)=	=A(4)
4	4	A(4)=	=A(5)

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.

Example-4 continued…

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

- The dependence distance is (1,-1)

  Iteration vector of sink – iteration vector of source.

  (3,2)-(2,3)=(1,-1)

  (3,3)-(2,4)=(1,-1)

  (4,2)-(3,3)=(1,-1)

  (4,3)-(3,4)=(1,-1)

- Direction vector (<,>)

		source	sink
I	J	S: a(i,j)	S: a(i-1,j+1)
2	2	a(2,2)	a(1,3)
2	3	a(2,3)	a(1,4)
2	4	a(2,4)	a(1,5)
3	2	a(3,2)	a(2,3)
3	3	a(3,3)	a(2,4)
3	4	a(3,4)	a(2,5)
4	2	a(4,2)	a(3,3)
4	3	a(4,3)	a(3,4)
4	4	a(4,4)	a(3,5)

Loop Carried Dependence

- Statement ***S2*** has a ***loop-carried dependence*** on statement ***S1***, **if and only if *S1*** references **location *M*** on iteration ***i***, ***S2*** references ***M*** on iteration ***j*** and ***d*(*i,j*) > 0** (that is, ***D*(*i,j*)** contains a "<" as leftmost non "=" component).

Example:

```
DO I = 1, N
S1        A(I+1) = F(I+1)
S2        F(I) = A(I)
ENDDO
```

- Array A: $S_1 \, \delta \, S_2$ dep. distance (1), direction vector (<)
- Array F: $S_1 \, \delta^{-1} \, S_2$ dep. distance (1), direction vector (<)

Loop Carried Dependence

- Level of a loop-carried dependence is the index of the leftmost non-“=” of D(i,j) for the dependence.

For instance:

```
DO I = 1, 10
   DO J = 1, 10
      DO K = 1, 10
S1       A(I, J, K+1) = A(I, J, K)
      ENDDO
   ENDDO
ENDDO
```

- Direction vector for S1 is (=, =, <)
- Level of the dependence is 3
- A level-k dependence between S_1 and S_2 is denoted by $S_1\, \delta_k\, S_2$

Loop Carried Transformations

Example:

```
        DO I = 1, 10
S1          A(I+1) = F(I)
S2          F(I+1) = A(I)
        ENDDO
```

Can be transformed or Not?

Will the dependencies be retained or inverted?

can be transformed to:

```
        DO I = 1, 10
S2          F(I+1) = A(I)
S1          A(I+1) = F(I)
        ENDDO
```

Loop-Independent Dependence

Definition: Statement **S2** has a ***loop-independent dependence*** on statement S1, **if and only if** there exist **two iteration vectors *i*** and ***j*** such that:

- Statement **S1** refers to memory location ***M*** on **iteration *i***, **S2** refers to ***M*** on iteration ***j***, and ***i == j***
- There is a control flow path from **S1** to **S2** within the iteration.

Example:

```
DO I = 1, 10
S1      A(I) = ...
S2      ... = A(I)
ENDDO
```

Loop-Independent Dependence

More complicated example:

```
DO I = 1, 9
S1        A(I) = ...
S2        ...   = A(10-I)
ENDDO
```

- S_1 stores and S_2 reads A(5) during iteration I=5
- all other dependences are loop carried

Parallelization and Vectorization

Theorem: it is valid to convert a sequential loop to a parallel loop *if the loop carries no dependence*.

- Want to convert loops like:

```
DO I=1,N
    X(I) = X(I) + C
ENDDO
```

- to `X(1:N) = X(1:N) + C` (Fortran 77 to Fortran 90)

- However:

```
DO I=1,N
    X(I+1) = X(I) + C
ENDDO
```

is not equivalent to `X(2:N+1) = X(1:N) + C`

Loop Distribution

Can statements in loops which carry dependences be vectorized?

```
      DO I = 1, N
S1        A(I+1) = B(I) + C
S2        D(I) = A(I) + E
      ENDDO
```

- Dependence: $S_1 \, \delta_1 \, S_2$ can be converted to:

```
S1    A(2:N+1) = B(1:N) + C
S2    D(1:N) = A(1:N) + E
```

Loop Distribution – Example2

```
      DO I = 1, N
S1        A(I+1) = B(I) + C
S2        D(I) = A(I) + E
        ENDDO
```

- **transformed to:**

```
    DO I = 1, N
S1      A(I+1) = B(I) + C
    ENDDO
    DO I = 1, N
S2      D(I) = A(I) + E
    ENDDO
```

- **leads to:**

```
S1      A(2:N+1) = B(1:N) + C
S2      D(1:N) = A(1:N) + E
```

Loop Distribution

Loop distribution fails if there is a cyclic dependence

```
   DO I = 1, N
S1     A(I+1) = B(I) + C
S2     B(I+1) = A(I) + E
   ENDDO
```

$S_1 \, \delta_1 \, S_2$ and $S_2 \, \delta_1 \, S_1$

Loop can be distributed
→ only 1 loop carried dependence

- What about:

```
   DO I = 1, N
S1        B(I) = A(I) + E
S2        A(I+1) = B(I) + C
   ENDDO
```

Dependence Testing Complications

- Un-known Loop bounds:

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

*What is the relationship between **N** and **10**?*

If **N<=10** → No loop carried dependence

If **N>10** → loop carried dependences

Dependence Testing Complications

- Triangular loops:

```
for (j = 0; j < height; j++) {
    for (i = 0; i <= j; i++) {
        printf("*");
    }
    printf("\n");
}
```

Example run:
height of triangle: **6**

```
*
**
***
****
*****
******
```

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

Dependence Testing Complications

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

i → 1 to 3

J → 1 to 2

i	j	A (I,j)	A(j,1)
1	1	A(1,1)	A(1,1)
1	2	A(1,2)	A(2,1)
2	1	A(2,1)	A(1,2)
2	2	A(2,2)	A(2,2)
3	1	A(3,1)	A(1,3)
3	2	A(3,2)	A(2,1)

Dependence Testing Complications

- User Variables:

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

- Serious Problem: *Aliases/Pointers*

Eliminating Anti and Output Dependence

- Anti- and output dependences can always be eliminated through variable renaming.

```
      DO I = 1, N
S1      A(I) = A(I+1)
      ENDDO
```

Anti dependence within loop

```
DO I = 1, N
  A2(I) = A(I)
ENDDO
```

Anti dependence across loops does not prevent loop parallelization

```
DO I = 1, N
  A(I) = A2(I+1)
ENDDO
```

Loop Parallelization

- A dependence is said to be carried by a loop, if the loop is the outermost 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
δ_{=,=}                 a(i, j)   =
                        ...               = a(i, j)

δ_{=,<}                 b(i, j)   =
                                  = b(i, j-1)

δ_{<,=}                 c(i, j)   =
                                  = c(i-1, j)
                     end do
                  end do
```

- Outermost loop with non “=“ dependence carries it.

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-1

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

- 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-2

```
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.**

Techniques for Breaking Dependencies and for Dealing with Scalars

- **Privatization**
  - **remove dependencies** created by use of **temporary workspaces**
- **Induction Variable Substitution**
  - find **closed solutions** for **basic induction variables**
- **Reduction**
  - Use **reductions** as compared to **own-implemented code** etc.

Privatization or Scalar Expansion

```
INTEGER J
DO I = 1, N
   J =
   A(I) = J
ENDDO
```

→

```
INTEGER J, Jx(N)
DO I = 1, N
   Jx(I) =
   A(I) = Jx(I)
ENDDO
J=Jx(N)
```

- All scalar assignments will cause loop-carried dependencies

- Can create local per-iteration or (more practically) per thread copies
  – **scalar expansion** or **privatization**

Induction Variable Substitution

```
DO I = 1, N
   J = J + K
   A(I) = J
ENDDO
```

⇨

```
DO I = 1, N
   A(I) = I*K
ENDDO
```

- Basic induction variables cause flow dependencies
- Can be replaced by a closed-form solution
  - an induction variable derived from the loop control variable

Reduction

```
DO I = 1, N
    sum = sum + A(I)
ENDDO
```

⇒ Reduction(ADD, A, N, sum)

Any Questions