Data Dependence and Vectorization

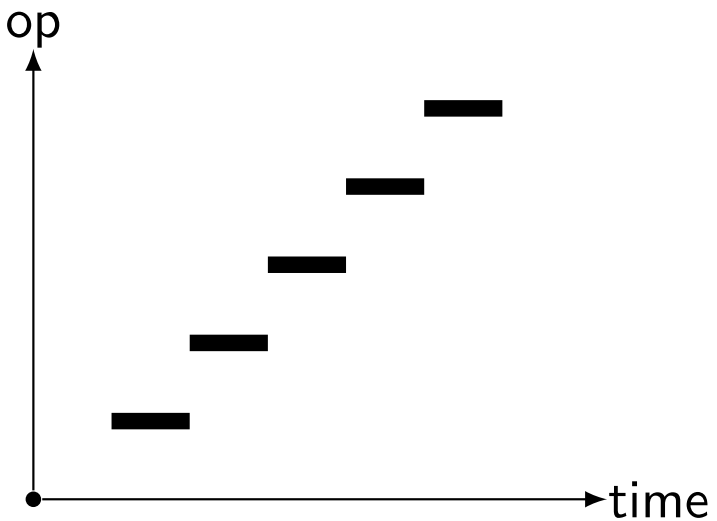
David Laukamp Martin Wallraff

July 23, 2014

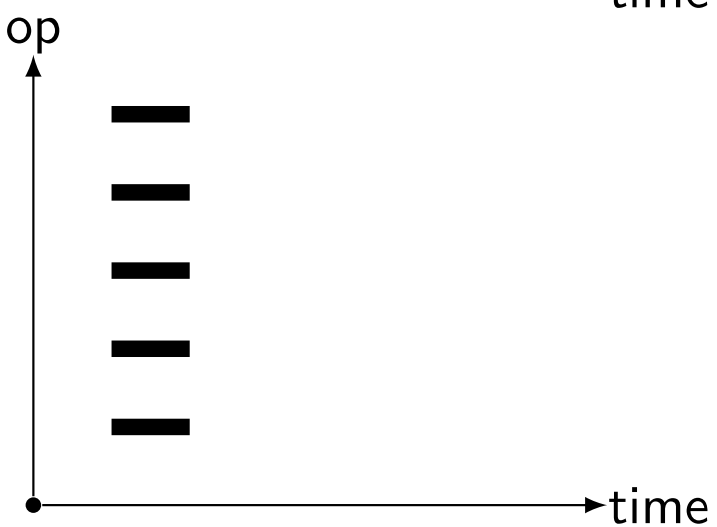
Contents

Vectorization - Basic Idea

```
DIMENSION Arr(5)
DO I=1,5
  Arr(I) = 0
END DO I
```

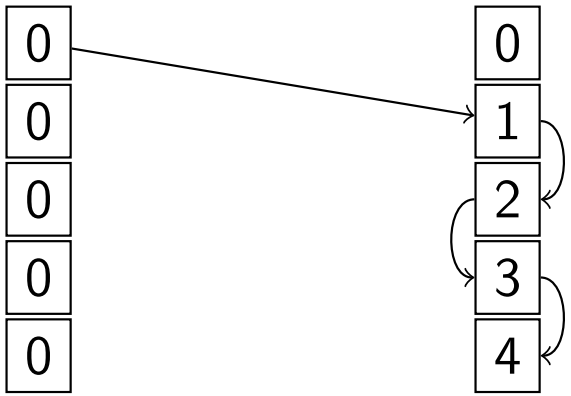



```
DIMENSION Arr(5)
Arr(1:5) = 0
```



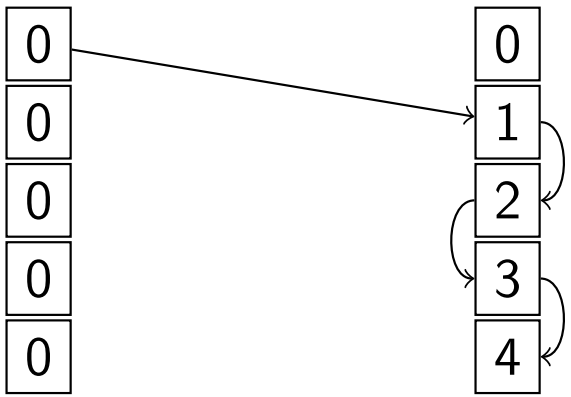
Problem

```
DIMENSION Arr(5)
DO I=1,4
  Arr(I+1) = Arr(I)+1
END DO I
```

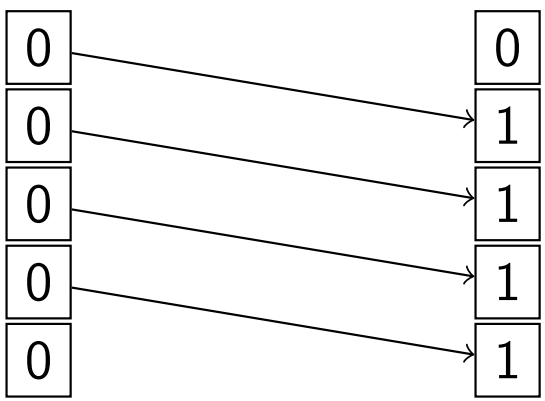


Problem

```
DIMENSION Arr(5)
DO I=1,4
  Arr(I+1) = Arr(I)+1
END DO I
```




```
DIMENSION Arr(5)
Arr(2:5) = Arr(1:4)+1
```



Contents

Program Model

Variables

```
INTEGER I, J
DIMENSION Arr(100),
          Brr(100,100)
```

Program Model

Variables

```
INTEGER  I , J
DIMENSION  Arr(100),
           Brr(100,100)
```

Statements

S_1 `I=I+1`
S_2 `Arr(I)=0`
S_3 `Brr(1,1:100)=Arr(1:100)`

Program Model

Variables

```
INTEGER I, J
DIMENSION Arr(100),
          Brr(100,100)
```

Do-Loops

```
DO I=0,100
 DO J=10,1,-1
 ...
 END DO J
END DO I
```

Statements

S_1 I=I+1
S_2 Arr(I)=0
S_3 Brr(1,1:100)=Arr(1:100)

Contents

Dependence Types

True dependence:

$$S_1 \delta^t S_2$$

$$\begin{array}{ll} S_1 & \mathsf{A} = \ldots \\ & \vdots \\ S_2 & \ldots = \mathsf{A} \end{array}$$

Dependence Types

True dependence:

$$S_1 \delta^t S_2$$

$$\begin{array}{ll} S_1 & \mathsf{A} = \ldots \\ & \vdots \\ S_2 & \ldots = \mathsf{A} \end{array}$$

Anti dependence:

$$S_1 \delta^a S_2$$

$$\begin{array}{ll} S_1 & \ldots = \mathsf{A} \\ & \vdots \\ S_2 & \mathsf{A} = \ldots \end{array}$$

Dependence Types

True dependence:

$S_1 \delta^t S_2$

$$\begin{array}{ll} S_1 & \mathsf{A} = \ldots \\ & \vdots \\ S_2 & \ldots = \mathsf{A} \end{array}$$

Anti dependence:

$S_1 \delta^a S_2$

$$\begin{array}{ll} S_1 & \ldots = \mathsf{A} \\ & \vdots \\ S_2 & \mathsf{A} = \ldots \end{array}$$

Output dependence:

$S_1 \delta^o S_2$

$$\begin{array}{ll} S_1 & \mathsf{A} = \ldots \\ & \vdots \\ S_2 & \mathsf{A} = \ldots \end{array}$$

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$
  - forward

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$
  - forward
  - loop-independent

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$
  - forward
  - loop-independent
- $S_2 \delta^t S_3$

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$
  - forward
  - loop-independent
- $S_2 \delta^t S_3$
  - forward

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$
  - forward
  - loop-independent
- $S_2 \delta^t S_3$
  - forward
  - loop-carried

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

Dependence Types (1)

$S\delta S'$ is a *forward*/*backward* dependence if S' is *textually* after/before S.

```
      DO I=1,100
S1      ... = Arr(I+2)
        :
        :
S2      Arr(I+2) = ...
        :
        :
S3      ... = Arr(I)
      END DO
```

- $S_1 \delta^a S_2$
  - forward
  - loop-independent
- $S_2 \delta^t S_3$
  - forward
  - loop-carried

$S\delta S'$ is *loop-carried* if S is executed in a different iteration than S'.

All loop-independent dependencies are forward.

Dependence Types (2)

```
      DO I=1,100
S1      ... = Arr(I)
        :
        :
S2      Arr(I+2) = ...
      END DO
```

Dependence Types (2)

```
      DO I=1,100
S1      ... = Arr(I)
        :
        :
S2      Arr(I+2) = ...
      END DO
```

- $S_2 \delta^t S_1$

Dependence Types (2)

```
      DO I=1,100
S1      ... = Arr(I)
        :
        :
S2      Arr(I+2) = ...
      END DO
```

- $S_2 \delta^t S_1$
  - backward

Dependence Types (2)

```
      DO I=1,100
S1      ... = Arr(I)
        :
        :
S2      Arr(I+2) = ...
      END DO
```

- $S_2 \delta^t S_1$
  - backward
  - loop-carried

Dependence Types (2)

```
      DO I=1,100
S1      ... = Arr(I)
        :
        :
S2      Arr(I+2) = ...
      END DO
```

- $S_2 \delta^t S_1$
  - backward
  - loop-carried

All backward dependencies are loop-carried.

Dependence Tests

How to test for dependence automatically?

Separability Test

```
      DO I=1,100
S1      Arr(3+2*I)=...
        :
        :
S2      ... =Arr(-1+2*I)
      END DO
```

Separability Test

```
      DO I=1,100
S1      Arr(3+2*I)=...
        :
        :
S2      ... =Arr(-1+2*I)
      END DO
```

Separability Condition

Equate subscript terms:

Separability Test

```
      DO  I=1,100
S1      Arr(3+2*I)=...
        :
        :
S2      ... =Arr(-1+2*I)
      END DO
```

Separability Condition

Equate subscript terms:

$$a_0 + a * x = b_0 + b * y, \quad x, y \in \{lb, ub\}$$

Integer solutions within $\{lb, ub\} \rightsquigarrow$ Dependence!

Separability Test

```
      DO  I=1,100
S1      Arr(3+2*I)=...
        :
        :
S2      ... =Arr(-1+2*I)
      END DO
```

$$3 + 2 * x = -1 + 2 * y$$

Separability Condition

Equate subscript terms:

$$a_0 + a * x = b_0 + b * y, \quad x, y \in \{lb, ub\}$$

Integer solutions within $\{lb, ub\} \rightsquigarrow$ Dependence!

Separability Test

```
      DO  I=1,100
S1      Arr(3+2*I)=...
        :
S2      ... =Arr(-1+2*I)
      END DO
```

$$\Rightarrow \qquad \begin{aligned} 3 + 2 * x &= -1 + 2 * y \\ y &= x + 2 \end{aligned}$$

Separability Condition

Equate subscript terms:

$$a_0 + a * x = b_0 + b * y, \quad x, y \in \{lb, ub\}$$

Integer solutions within $\{lb, ub\} \rightsquigarrow$ Dependence!

Separability Test: Overview

Advantages:

Disadvantages:

Separability Test: Overview

Advantages:

- necessary and sufficient criteria for dependence

Disadvantages:

Separability Test: Overview

Advantages:

- necessary and sufficient criteria for dependence
- computationally efficient

Disadvantages:

Separability Test: Overview

Advantages:

- necessary and sufficient criteria for dependence
- computationally efficient
- often applicable

Disadvantages:

Separability Test: Overview

Advantages:

- necessary and sufficient criteria for dependence
- computationally efficient
- often applicable

Disadvantages:

- very expensive for multiple index vars

Separability Test: Overview

Advantages:

- necessary and sufficient criteria for dependence
- computationally efficient
- often applicable

Disadvantages:

- very expensive for multiple index vars

Consequence: Only used on single index vars, for multiple use gcd test.

gcd Test

```
   DO I=1,10
    DO J=1,5
S1    Arr(3I+6J-8) = ...
            :
            :
S2    ... = Arr(3I-3J+33)
    END DO J
   END DO I
```

gcd Test

```
   DO  I=1,10
    DO  J=1,5
S1    Arr(3I+6J-8) = ...
            :
S2    ... = Arr(3I-3J+33)
    END DO  J
   END DO  I
```

Build Diophantine equation:

$$3x_I + 6x_J - 3y_I + 3y_J = 25$$

gcd Test

```
   DO I=1,10
    DO J=1,5
S1    Arr(3I+6J-8) = ...
            :
            :
S2    ... = Arr(3I-3J+33)
    END DO J
   END DO I
```

Build Diophantine equation:

$$3x_I + 6x_J - 3y_I + 3y_J = 25$$

Apply gcd test:

$gcd(3, 6, -3, 3) | 25 \rightarrow$ Independent!

gcd Test

```
   DO I=1,10
    DO J=1,5
S1    Arr(3I+6J-8) = ...
          :
S2    ... = Arr(3I-3J+33)
    END DO J
   END DO I
```

```
   DO I=1,10
    DO J=1,5
S1    Arr(I+2J) = ...
          :
S2    ... = Arr(I-3J+33)
    END DO J
   END DO I
```

Build Diophantine equation:

$$3x_I + 6x_J - 3y_I + 3y_J = 25$$

Apply gcd test:

$gcd(3, 6, -3, 3) | 25 \rightarrow$ Independent!

gcd Test

```
   DO I=1,10
    DO J=1,5
S1    Arr(3I+6J-8) = ...
          :
S2    ... = Arr(3I-3J+33)
    END DO J
   END DO I
```

```
   DO I=1,10
    DO J=1,5
S1    Arr(I+2J) = ...
          :
S2    ... = Arr(I-3J+33)
    END DO J
   END DO I
```

Build Diophantine equation:

$3x_I + 6x_J - 3y_I + 3y_J = 25$ $\qquad$ $x_I + 2x_J - y_I + 3y_J = 33$

Apply gcd test:

$gcd(3, 6, -3, 3) | 25 \rightarrow$ Independent!

gcd Test

```
   DO I=1,10
    DO J=1,5
S1    Arr(3I+6J-8) = ...
          :
S2    ... = Arr(3I-3J+33)
    END DO J
   END DO I
```

```
   DO I=1,10
    DO J=1,5
S1    Arr(I+2J) = ...
          :
S2    ... = Arr(I-3J+33)
    END DO J
   END DO I
```

Build Diophantine equation:

$3x_I + 6x_J - 3y_I + 3y_J = 25$ $\qquad$ $x_I + 2x_J - y_I + 3y_J = 33$

Apply gcd test:

$gcd(3, 6, -3, 3) | 25 \rightarrow$ Independent! $\qquad$ $gcd(1, 2, -1, 3) | 33 \rightarrow$ Dependence?

gcd Test: Overview

Advantages:

Disadvantages:

gcd Test: Overview

Advantages:

- computationally efficient

Disadvantages:

gcd Test: Overview

Advantages:

- computationally efficient
- applicable for multiple index vars

Disadvantages:

gcd Test: Overview

Advantages:

- computationally efficient
- applicable for multiple index vars

Disadvantages:

- no sufficient criteria for dependence

gcd Test: Overview

Advantages:

- computationally efficient
- applicable for multiple index vars

Disadvantages:

- no sufficient criteria for dependence
- in practice, the gcd is frequently 1

gcd Test: Overview

Advantages:

- computationally efficient
- applicable for multiple index vars

Disadvantages:

- no sufficient criteria for dependence
- in practice, the gcd is frequently 1

$\Rightarrow$ no useful result

Contents

Vectorization: Try & Error

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

Vectorization: Try & Error

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Vectorization: Try & Error

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

Vectorization: Try & Error

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO  I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

S_1 Arr(1:10)=Brr(1:10)
S_2 Crr(1:10)=Arr(2:11)

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

S_1 $Arr(1:10)=Brr(1:10)$
S_2 $Crr(1:10)=Arr(2:11)$

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

S_1 $Arr(1:10)=Brr(1:10)$
S_2 $Crr(1:10)=Arr(1:10)$

Vectorization: Try & Error

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO  I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(1:10)
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(1:10)
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

Why?

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(1:10)
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

Why?

$S_2 \delta S_1$: loop-carried

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

$S_2 \delta S_1$: loop-carried

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(1:10)
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

Why?

$S_1 \delta S_2$: loop-independent

Vectorization: Try & Error

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I)
   END DO I
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(1:10)
```

Crr(I):	Brr(1)	Brr(2)	Brr(3)	...

Why?

$S_2 \delta S_1$: loop-carried

$S_1 \delta S_2$: loop-independent

All loop-independent dependencies can be directly vectorized!

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Statement Re-Ordering

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Statement Re-Ordering

```
   DO  I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO  I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Crr(1:10)=Arr(2:11)
S2   Arr(1:10)=Brr(1:10)
```

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
S1   Crr(1:10)=Arr(2:11)
S2   Arr(1:10)=Brr(1:10)
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
S1   Crr(1:10)=Arr(2:11)
S2   Arr(1:10)=Brr(1:10)
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Why?

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1   Arr(1:10)=Brr(1:10)
S2   Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
S1   Crr(1:10)=Arr(2:11)
S2   Arr(1:10)=Brr(1:10)
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Why?

$S_2 \delta S_1$: loop-carried backward

Statement Re-Ordering

```
   DO I=1,10
S1   Arr(I)=Brr(I)
S2   Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S2   Crr(I)=Arr(I+1)
S1   Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1  Arr(1:10)=Brr(1:10)
S2  Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
S1  Crr(1:10)=Arr(2:11)
S2  Arr(1:10)=Brr(1:10)
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Why?

$S_2 \delta S_1$: loop-carried backward

$S_2 \delta S_1$: loop-carried forward

Statement Re-Ordering

```
   DO I=1,10
S1 Arr(I)=Brr(I)
S2 Crr(I)=Arr(I+1)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
   DO I=1,10
S2 Crr(I)=Arr(I+1)
S1 Arr(I)=Brr(I)
   END DO I
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

```
S1 Arr(1:10)=Brr(1:10)
S2 Crr(1:10)=Arr(2:11)
```

Crr(I):	Brr(2)	Brr(3)	Brr(4)	...

```
S1 Crr(1:10)=Arr(2:11)
S2 Arr(1:10)=Brr(1:10)
```

Crr(I):	Arr(2)	Arr(3)	Arr(4)	...

Why?

$S_2 \delta S_1$: loop-carried backward

$S_2 \delta S_1$: loop-carried forward

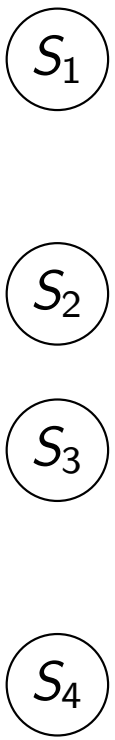
All loop-carried forward dependent statements can be vectorized!

Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```

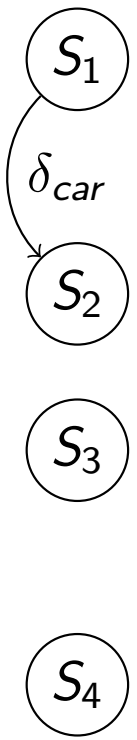
Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```



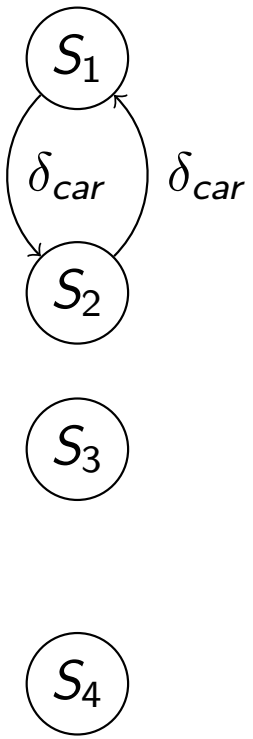
Vector Statement Generation (1)

```
   DO  I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO  I
```



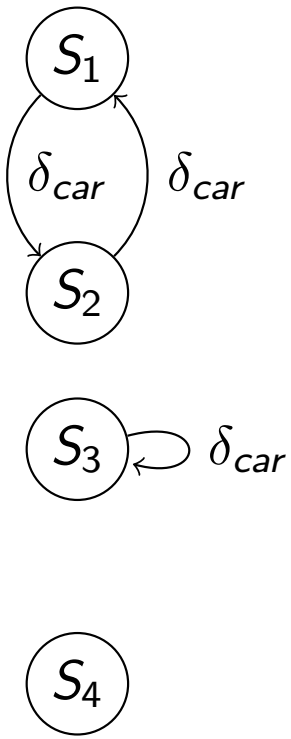
Vector Statement Generation (1)

```
   DO  I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO  I
```



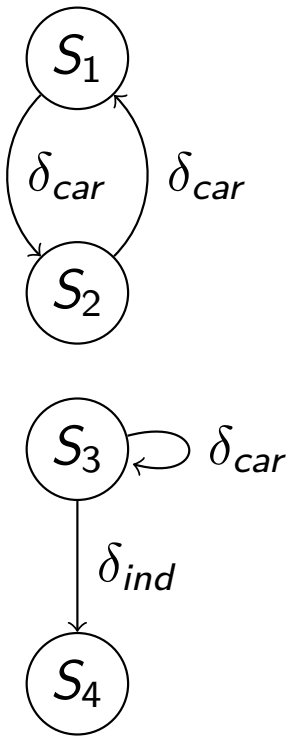
Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```



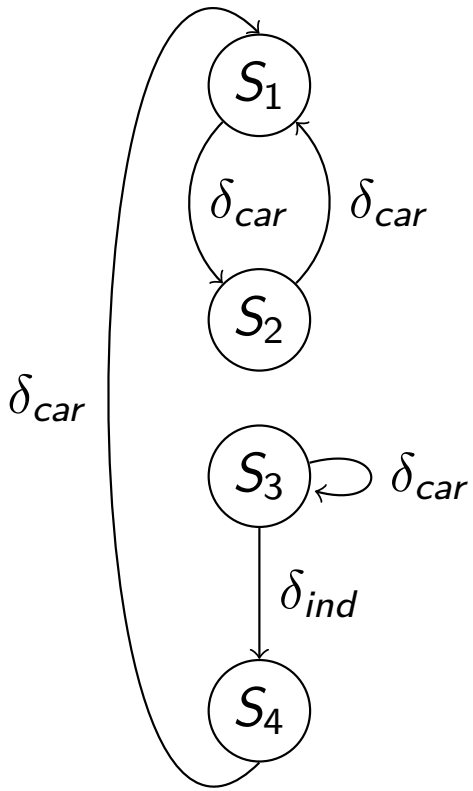
Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```



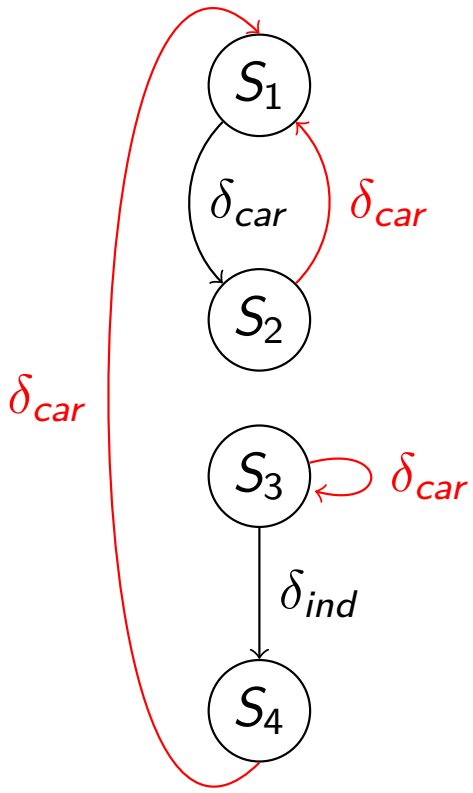
Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```



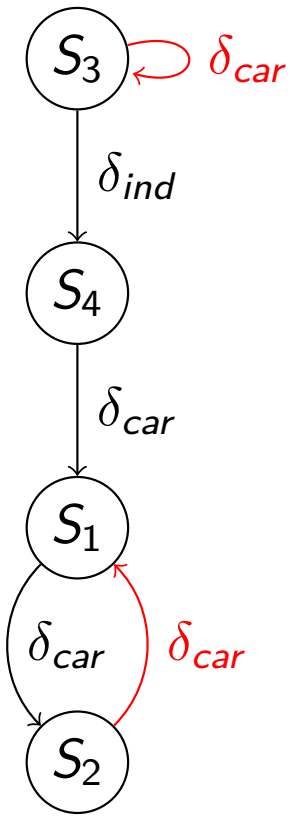
Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```



Vector Statement Generation (1)

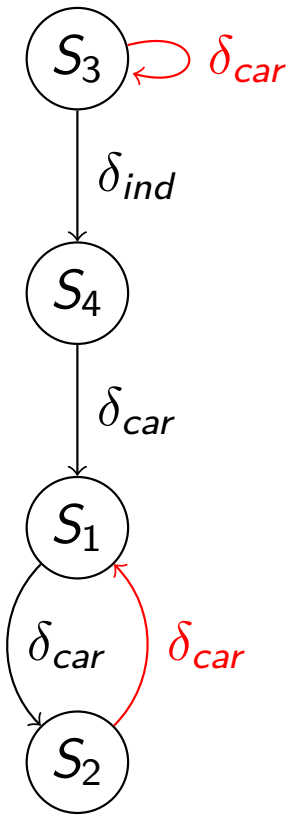
```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```



Vector Statement Generation (1)

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
   END DO I
```

```
   DO I=2,100
S3   Crr(I)=Crr(I-1)+3
S4   Drr(I)=Crr(I)
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
   END DO I
```

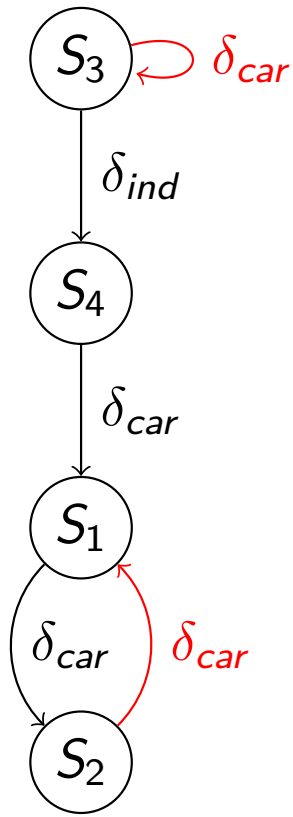


Vector Statement Generation (2)

```
   DO I=2,100
S3   Crr(I)=Crr(I-1)+3
   END DO I
```

```
S4 Drr(2:100)=Crr(2:100)
```

```
   DO I=2,100
S1   Arr(I)=Drr(I-1)+Brr(I-1)
S2   Brr(I)=Arr(I-1)*2
   END DO I
```



Partial Vectorization

```
   DO I=0,100
    DO J=0,100
S1    Arr(I,J) = 3 + Arr(I-1,J)
    END DO J
   END DO I
```

Partial Vectorization

```
   DO I=0,100
    DO J=0,100
S1    Arr(I,J) = 3 + Arr(I-1,J)
    END DO J
   END DO I
```

```
   DO I=0,100
S1  Arr(I,0:100) = 3 + Arr(I-1,0:100)
   END DO I
```

Contents

Do-Loop Index Normalization

```
DO I=10,0,-2
  Arr(I) = ...
END DO I
```

Do-Loop Index Normalization

Normalization steps:

1. introduce new variables

```
DO  I=10,0,-2
  Arr(I) = ...
END DO  I
```

```
DO  Î=1,(0-10+-2)/-2      |= 6
  Arr(10+(Î-1)*-2) = ...
END DO  Î
Î=10+MAX(6,0)*-2          |= -2
```

Do-Loop Index Normalization

Normalization steps:

1. introduce new variables
2. replace all occurrences of original index variables

```
DO  I=10,0,-2
  Arr(I) = ...
END DO  I
```

```
DO  Î=1,(0-10+-2)/-2      |= 6
  Arr(10+(Î-1)*-2) = ...
END DO  Î
Î=10+MAX(6,0)*-2          |= -2
```

Do-Loop Index Normalization

Normalization steps:

1. introduce new variables
2. replace all occurrences of original index variables
3. iterate from 1 to N in steps of 1

```
DO I=10,0,-2
  Arr(I) = ...
END DO I
```

```
DO Î=1,(0-10+-2)/-2     |=6
  Arr(10+(Î-1)*-2) = ...
END DO Î
Î=10+MAX(6,0)*-2         |=-2
```

Do-Loop Index Normalization

Normalization steps:

1. introduce new variables
2. replace all occurrences of original index variables
3. iterate from 1 to N in steps of 1
4. ensure that postcondition is fulfilled

```
DO  I=10,0,-2
  Arr(I) = ...
END DO  I
```

```
DO  Î=1,(0-10+-2)/-2      |= 6
  Arr(10+(Î-1)*-2) = ...
END DO  Î
Î=10+MAX(6,0)*-2          |= -2
```

Scalar Forward Substitution

```
DO I=1,10
  g=4+3*I
  Arr(g) = ...
END DO I
```

Scalar Forward Substitution

Situation:

- non-index var g is used as subscript

```
DO  I=1,10
  g=4+3*I
  Arr(g) = ...
END DO  I
```

Scalar Forward Substitution

Situation:

- non-index var g is used as subscript
- g depends only on index var

```
DO  I=1,10
  g=4+3*I
  Arr(g) = ...
END DO  I
```

Scalar Forward Substitution

Situation:

- non-index var g is used as subscript
- g depends only on index var

Solution:

- replace uses of g by its definition

```
DO I=1,10
  g=4+3*I
  Arr(g) = ...
END DO I
```

```
DO I=1,10
  g=4+3*I
  Arr(4+3*I) = ...
END DO I
```

Induction Variable Substitution

```
INTEGER g=5
DO I=1,10
  g=g+3
  Arr(g+4) = ...
END DO I
```

Induction Variable Substitution

Situation:

- non-index var g is used as induction var

```
INTEGER g=5
DO I=1,10
  g=g+3
  Arr(g+4) = ...
END DO I
```

Induction Variable Substitution

Situation:

- non-index var g is used as induction var
- g's definition is of the form: $g = g + c$

```
INTEGER g=5
DO I=1,10
  g=g+3
  Arr(g+4) = ...
END DO I
```

Induction Variable Substitution

Situation:

- non-index var g is used as induction var
- g's definition is of the form: $g = g + c$

Solution:

- replace uses of g by: $g_{base} + c * I$

```
INTEGER g=5
DO I=1,10
  g=g+3
  Arr(g+4) = ...
END DO I
```

```
INTEGER g=5
DO I=1,10
  g=g+3
  Arr(5+3*I+4) = ...
END DO I
```

References


D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe.
Dependence graphs and compiler optimizations.
In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.


H. Zima and B. Chapman.
Supercompilers for Parallel and Vector Computers.
ACM, New York, NY, USA, 1991.