Automatic Parallelization Prospects

A Few Sources

- Michael Wolfe and Utpal Banerjee, Data Dependence and Its Application to Parallel Processing, Int'l J. of Parallel Programming, 16, 2, 1987, 137-178.
- Hans Zima, with Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1990.
- Thomas Bräunl, *Parallel Programming, an Introduction*, Prentice-Hall, 1993.
- Michael Wolfe, *High-Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.
- Kevin Dowd and Charles Severance, *High Performance Computing,* 2nd ed. O'Reilly, 1998.

Basic Idea

- By examining the data and control flow dependences in a program, it can be determined whether or not operations can be done in parallel.
- Such analysis can be be incorporated into compilers, for example, or into hardware itself.
- Even if not constructing a compiler, insights can be useful for algorithms and code.

Early Realizations in Hardware

- CDC 6600 instruction look-ahead
- IBM 360/91 data reservations and forwarding
- Texas Instruments ASC pipelining
- Cray-1 vector chaining

Vector Processor → GPGPU

- Initially graphics processor concentrated on vector operations.
- As more demands for flexibility and performance have been made, the general purpose GPU evolved.
- So some vector processing techniques can apply to newer GPU implementations.

Hardware vs. Software

- Hardware is easier, because there are fewer possible operations, but
- Hardware can only optimize on a local basis.
- A software view is required to get the big picture of optimization possibilities.

Target Constructs

- Ideally we have some target parallel constructs in which to transform our ordinary sequential source.

- Fortran-inspired examples are often used:
  - doall
  - doacross
  - forall
  - Fortran 90 array statements

Sequential DO

- Fortran Sequential DO:

```
do I = L, U, S        [lower, upper, stride]
        .
        .
        .
end
```

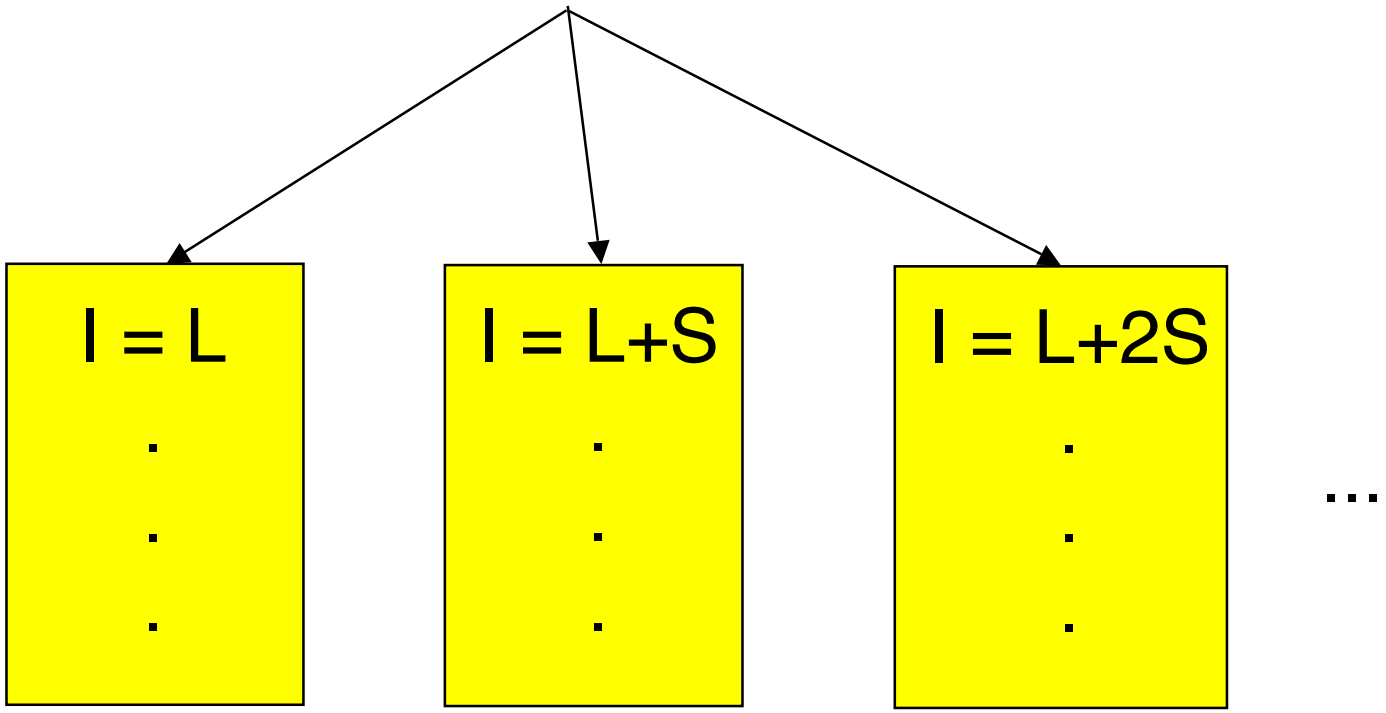
- Loop index runs I = L, L+S, L+2S, …, U

Parallel doall

doall I = L, U, S
[lower, upper, stride]

- Loop index runs I = L, L+S, L+2S, …, U
- All iterations executable in parallel
- Maintain sequentiality **within** a given iteration

doall I = L, U, S



doall Example

```
doall I = 1, 100
        A(I) = C*A(I)
        B(I) = A(I)
        D(I) = X*B(I)
end
```

Parallel across iterations,
sequential within each iteration

doall possibly non-deterministic (Wolfe&Banerjee)

Data dependence is a relation that *requires* execution of one statement before another; the *doall* semantics allow but do not require this execution ordering, so even though the data value may pass from one iteration to another, this is not a dependence. Thus, in a concurrent loop (in the absence of explicit synchronization), the only direction possible is the equal direction. The type of nondeterministic behavior allowed by our simplistic language definition may not be desirable in the applications world. An alternate definition would disallow nondeterminism in the

doacross

- In contrast to **doall**, dependencies in **doacross** *can* exist between iterations.
- Synch point and dependency distances control **synchronization**.

Doacross Example

```
doacross I = 1, 100
        await(1, 3, A(I-3))                inserted
        A(I) = B(I) + A(I-3)
        advance(1, A(I))                   inserted
end
```

The current iteration I is held at **await** until iteration I-3 reaches **advance**.

1 is the **Synch Point** number.

FORTRAN DOACROSS
Parallel Loop Construct

src: http://impact.crhc.illinois.edu/archives/ece412/public_html/Notes/412_lec22/sld011.htm

- Dependences exist between loop iterations.
- Dependences are enforced by two constructs:
  - **advance(synch_pt)** signals that the current iteration has passed the synchronization point identified by synch_pt.
  - **await(synch_pt, depend_distance)** forces the execution of the current iteration to wait for a previous iteration to pass the synchronization point identified by synch_pt. The iteration is the current iteration number minus depend_distance.

An OpenMP doacross

10.8 DOACROSS

src http://www.nersc.gov/nusers/resources/PDSF/documentation/pgi/pgiws_ug/pgi30u11.htm

The C$DOACROSS directive is not part of the OpenMP standard, but is supported for compatibility with programs parallelized using legacy SGI-style directives.

Syntax:

```
C$DOACROSS [ Clauses ]
< Fortran DO loop to be executed in parallel >
```

Clauses:

```
[ {PRIVATE | LOCAL} (list) ]
[ {SHARED | SHARE} (list) ]
[ MP_SCHEDTYPE={SIMPLE | INTERLEAVE} ]
[ CHUNK=<integer_expression> ]
[ IF (logical_expression) ]
```

The C$DOACROSS directive has the effect of a combined parallel region and parallel DO loop applied to the loop immediately following the directive. It is very similar to the OpenMP PARALLEL DO directive, but provides for backward compatibility with codes parallelized for SGI systems prior to the OpenMP standardization effort. The C$DOACROSS directive must not appear within a parallel region. It is a short-hand notation which tells the compiler to parallelize the loop to which it applies, even though that loop is not contained within a parallel region. While this syntax is more convenient, it should be noted that if multiple successive DO loops are to be parallelized it is more efficient to define a single enclosing parallel region and parallelize each loop using the OpenMP DO directive.

A variable declared PRIVATE or LOCAL to a C$DOACROSS loop is treated the same as a private variable in a parallel region or DO (see above). A variable declared SHARED or SHARE to a C$DOACROSS loop is shared among the threads, meaning that only 1 copy of the variable exists to be used and/or modified by all of the threads. This is equivalent to the default status of a variable that is not listed as PRIVATE in a parallel region or DO (this same default status is used in C$DOACROSS loops as well).

doacross vs. doall

- If a **doacross** has no inter-iteration dependencies, it becomes equivalent to a (deterministic) **doall**.

- But the two are different in the general case.

Fortran 90 and HPF Array Statements

- HPF = “High Performance Fortran”

- A(I:J) = B(I:J)
- A(I:J) = B(J:I-1) [reverses order]
- A(P) = B [P is an index vector]
- A = SQRT(B)+5. [map]

Vector forall

- Fortran-8X and Wolfe&Banerjee define this as vector execution of an index set:

  forall I=1, N A(I) = B(I) + C(I)

  means: that
  - all values of B(I) and C(I) are fetched,
  - all additions are done, then
  - all values of A(I) are stored.

forall vs. do

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

has a different meaning than either

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

or

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

Multi-Statement Forall

- forall i = 1, N
  A(I) = B(I)
  C(I) = A(I) + D(I)
  end

Statements are **sequential** within iteration.
Iterations are parallel.

Single statements executed as if a forall.

multi-statement forall example

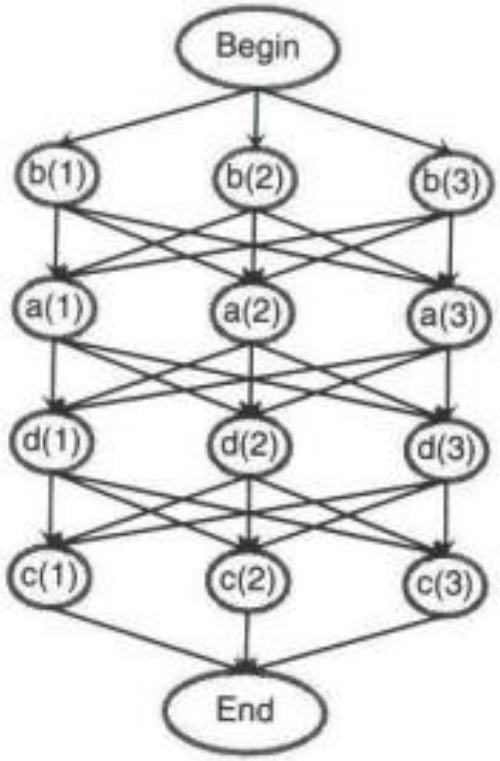


Figure 6.5
Precedence graph for a FORALL statement

```
FORALL (I = 1:3)
  a(I) = b(I)
  c(I) = d(I)
END FORALL
```

vs.multi-statement do

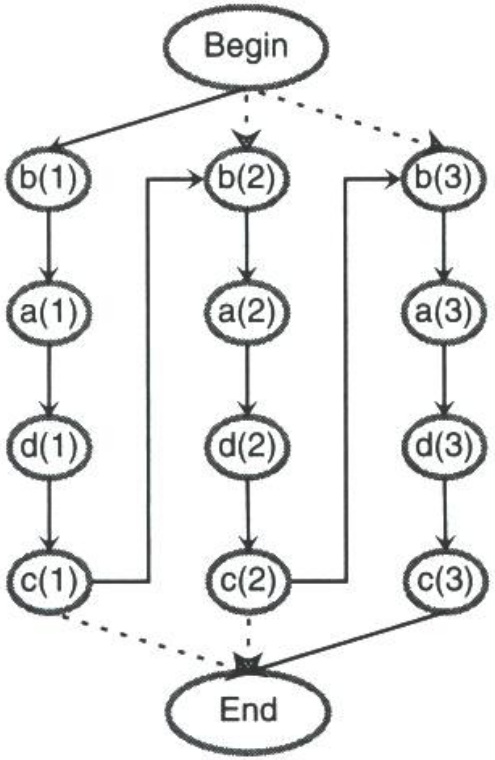
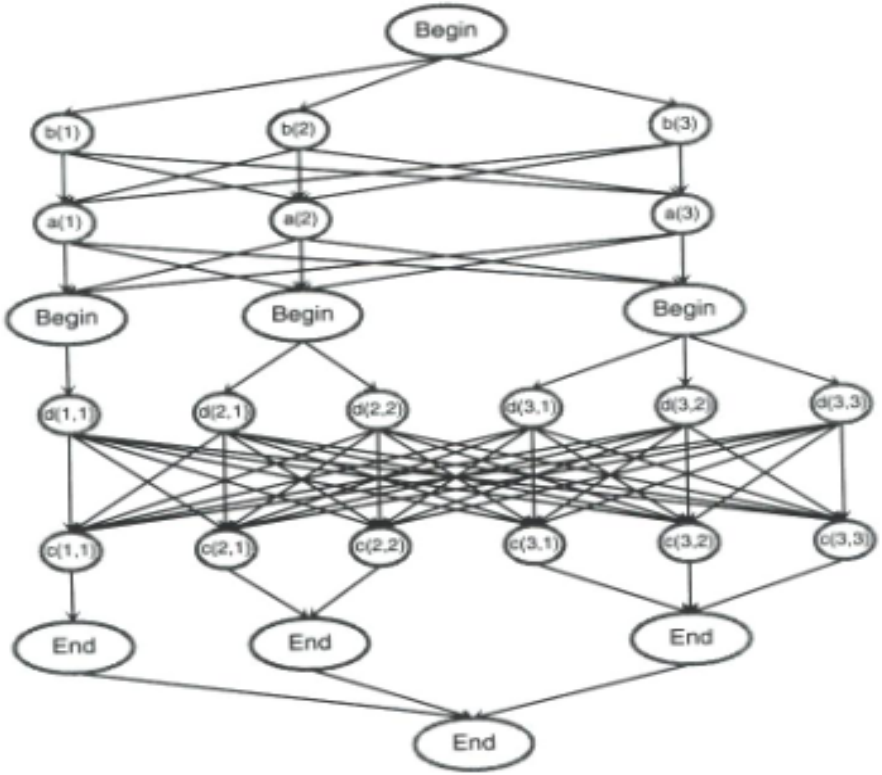


Figure 6.6
Precedence graph for a DO statement

```
DO I = 1, 3
  a(I) = b(I)
  c(I) = d(I)
END DO
```

nested forall example




```
FORALL (I = 1:3)
  a(I) = b(I)
  FORALL (J = 1:I)
    c(I,J) = d(I,J)
  END FORALL
END FORALL
```

Figure 6.7
Precedence graph for nested FORALL statements

Restrictions on forall nesting

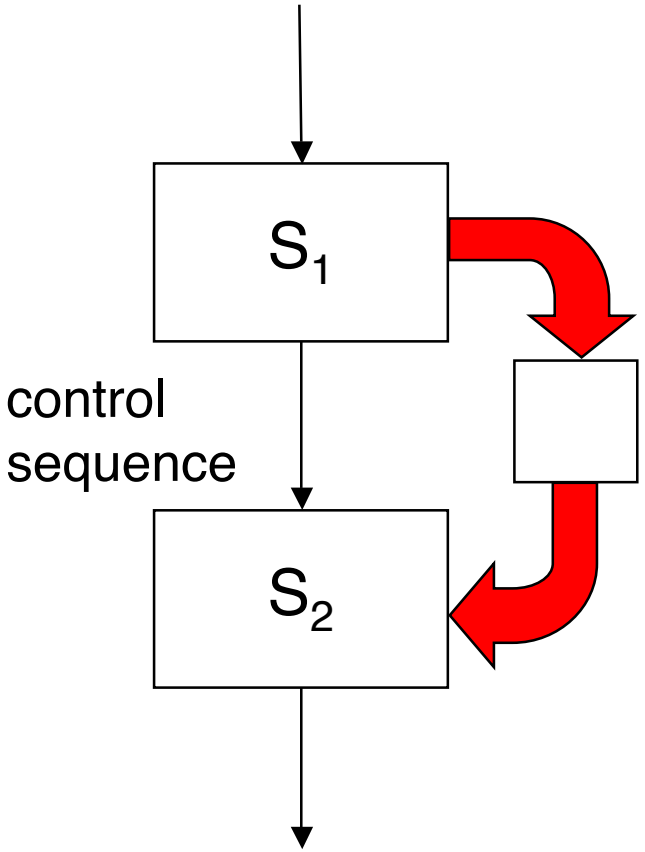
- Only other forall's can be nested inside a forall; not doall's nor doacross's.

Dependency Analysis

Bernstein's Conditions (1966) Scalar Considerations

- For a statement S:
  - IN(S) = set of variables, registers, or locations used by S
  - OUT(S) = set written to by S
- S_1; S_2 (sequence) is equivalent to S_1 II S_2 (parallel) **provided** that
  - $OUT(S_1) \cap OUT(S_2) = \varnothing$
  - $OUT(S_1) \cap IN(S_2) = \varnothing$
  - $OUT(S_2) \cap IN(S_1) = \varnothing$

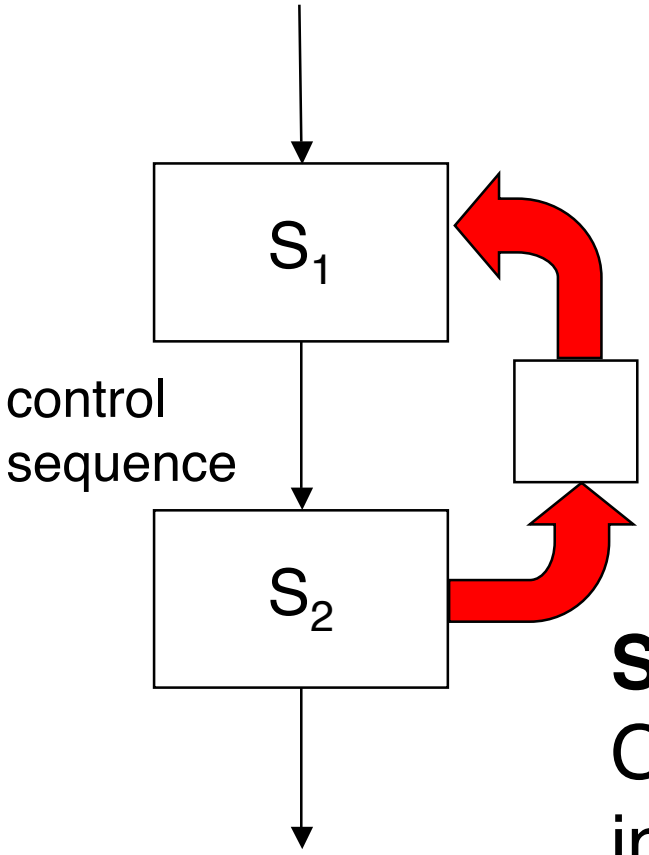
Condition Violations



$OUT(S_1) \cap IN(S_2) \neq \varnothing$

Essential Sequencing

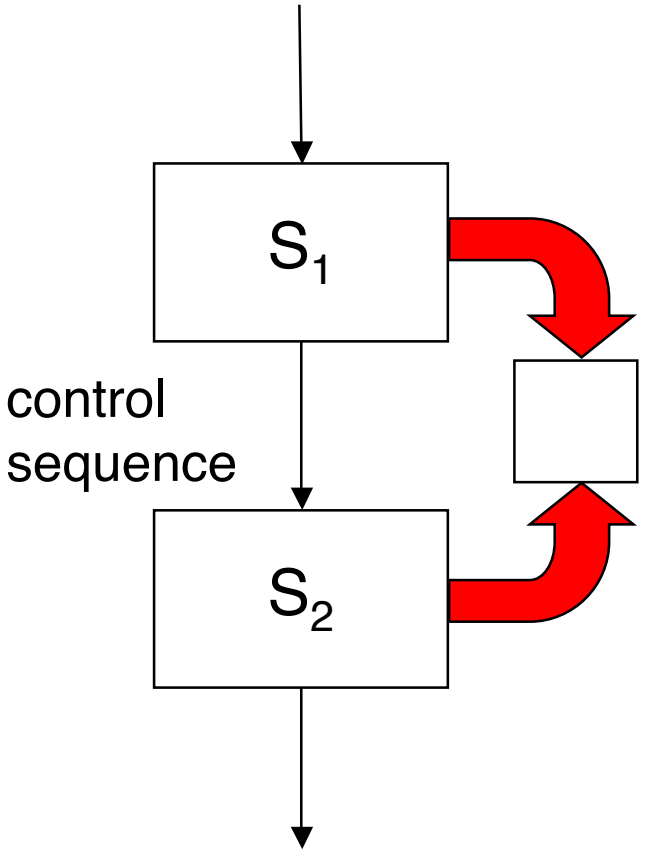
Condition Violations



$OUT(S_2) \cap IN(S_1) \neq \varnothing$

Sequencing, but Inessential:
$OUT(S_2)$ could be used by the ***next*** instance of S_1. The **current** instances of S_1 and S_2 could be concurrent.

Condition Violations



$OUT(S_1) \cap OUT(S_2) \neq \varnothing$

Inessential Sequencing:

$OUT(S_1)$ replaced. The value produced by S_2 could be put somewhere else.

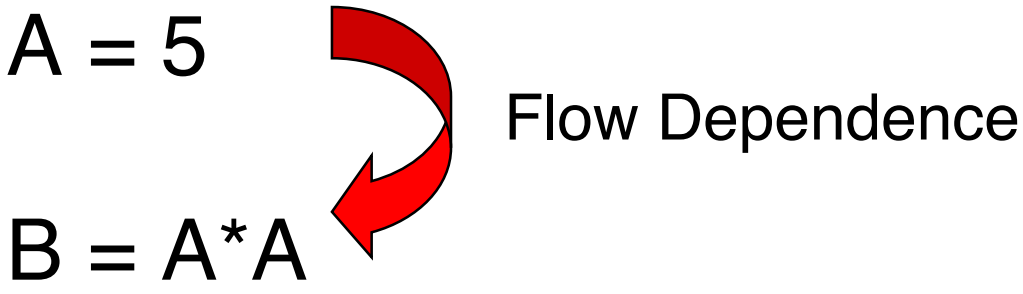
Data Dependence Classification

- Expresses constraints on parallel execution, as derived from sequential execution semantics

- Types of Dependence (Kuck, Wolfe, et al.):
  - Flow dependence
  - Anti dependence
  - Output dependence

Flow Dependence

- A variable set in one statement is used in a later one:

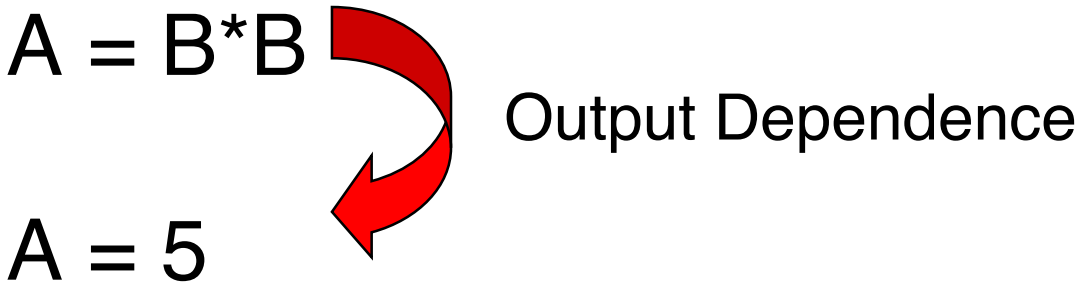


Anti Dependence

- A variable used in one statement is set in a later one:

Output Dependence

- A variable set in one statement is later set:

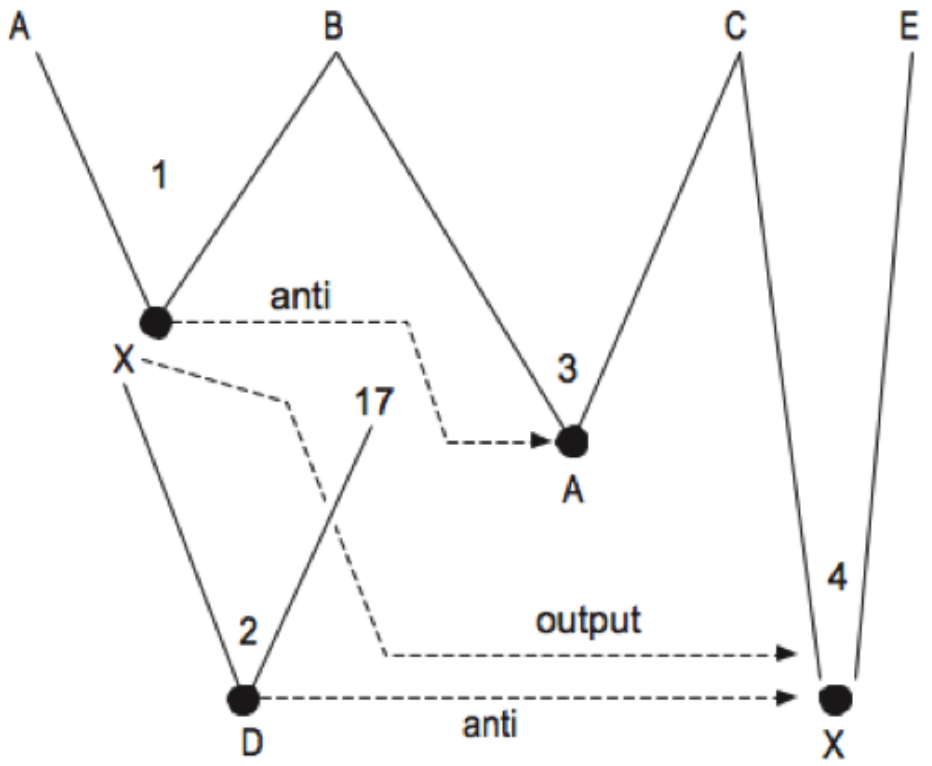


Combinations

- Shown before are just the bare possibilities.
- There can be more than one dependency involved in any pair of statements.

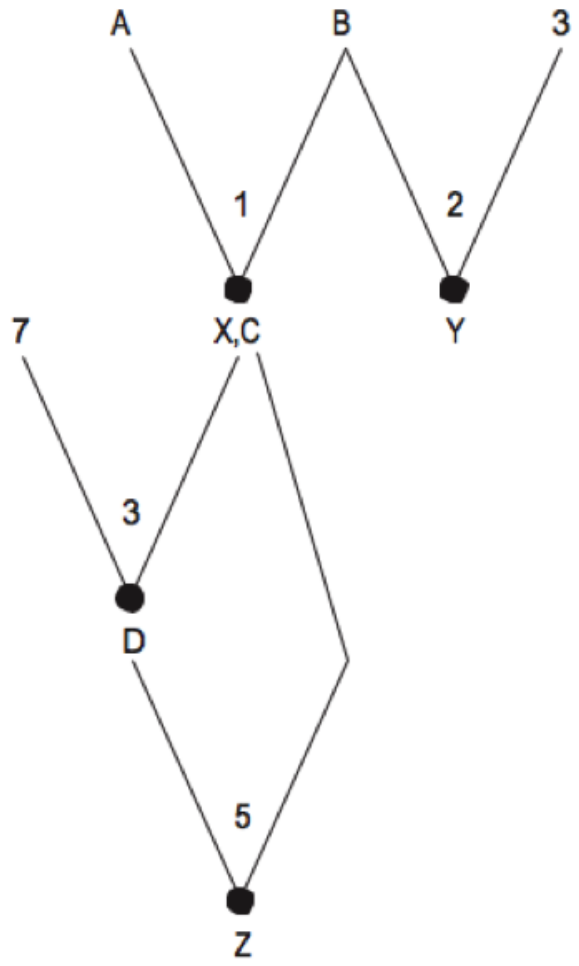
A Complex of Dependencies

X = A + B
D = X * 17
A = B + C
X = C + E



Parallel Execution Possibilities

X = A + B
Y = B + 3
D = X * 7
C = A + B
Z = D + C



Undecidability

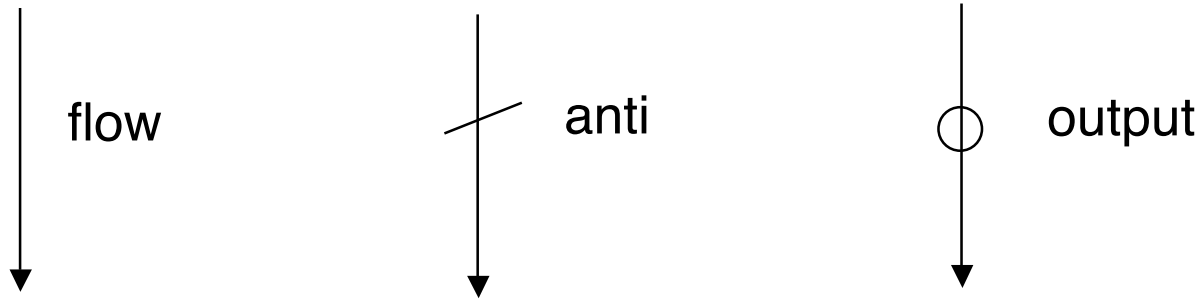
- There can be no algorithm that gives a total assessment of whether a dependency gets exercised for arbitrary programs. (Bernstein observed this.)
- Therefore we cannot construct a perfect parallelism analyzer.
- We have to settle for **sufficient** conditions for parallelism.

Removable Dependences

- Anti Dependence and Output Dependence are **removable.**
- They are artifacts of using variables as if memory location, rather than purely for their values.
- Flow Dependence is **not removable**; it expresses essential precedence.
- Clarification of whether location- or value-based dependency is being considered will be left to context.

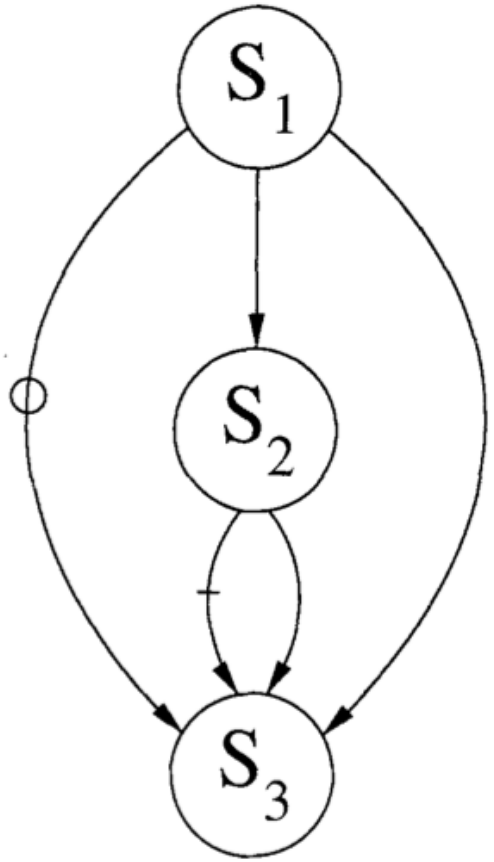
Notation (Bräunl, after Wolfe, et al.)

- $S_1 \delta^f S_2$ means S_2 is flow dependent on S_1
- $S_1 \delta^a S_2$ means S_2 is anti dependent on S_1
- $S_1 \delta^o S_2$ means S_2 is output dependent on S_1



Example (Wolfe & Banerjee)

$S_1: A = X + B$
$S_2: C = A * 3$
$S_3: A = A + C$



$\text{IN}(S_1) = \{X, B\}$ $\quad \text{OUT}(S_1) = \{A\}$
$\text{IN}(S_2) = \{A\}$ $\quad \text{OUT}(S_2) = \{C\}$
$\text{IN}(S_3) = \{A, C\}$ $\quad \text{OUT}(S_3) = \{A\}$

Non-Specific Dependency

Let $S_1 \delta\ S_2$ (without superscript) mean

$S_1 \delta^f\ S_2$

or

$S_1 \delta^a\ S_2$

or

$S_1 \delta^o\ S_2$

Indirect Dependency (Δ)

- Let $S_i \Delta S_j$ mean (recursively)

  $S_i \delta S_j$

  or for some S_k, $S_i \Delta S_k$ and $S_k \delta S_j$

In other words, Δ is the **transitive closure** of δ.

Execution Order (Wolfe)

- Let

  $S_1 \Theta S_2$

  mean S_1 is executed before S_2 in the original execution order of the program.

Example

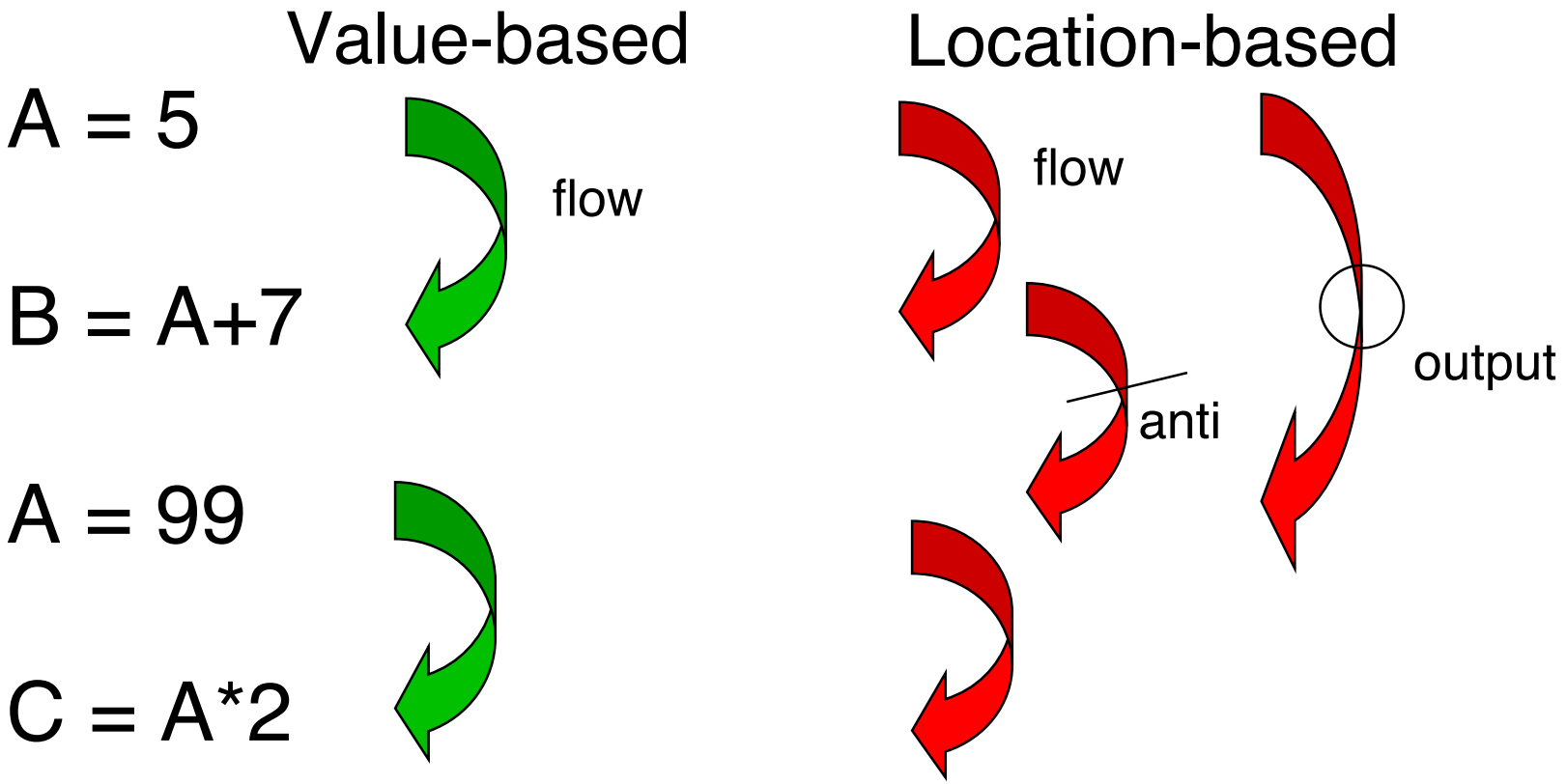
$$S_1: A = B + D$$
$$S_2: C = A * 3$$
$$S_3: A = A + C$$
$$S_4: E = A/2$$

$$S_1 \, \delta \, S_2$$
$$S_2 \, \delta \, S_3$$
$$S_3 \, \delta \, S_4$$

Note that $S_1 \, \delta \, S_4$ does not hold, even though $S_1 \, \Theta \, S_4$ and $\mathrm{OUT}(S_1) \cap \mathrm{IN}(S_4) \neq \emptyset$, because S_4 is not to use the value of A computed in S_1, but rather the value of A computed in S_3.

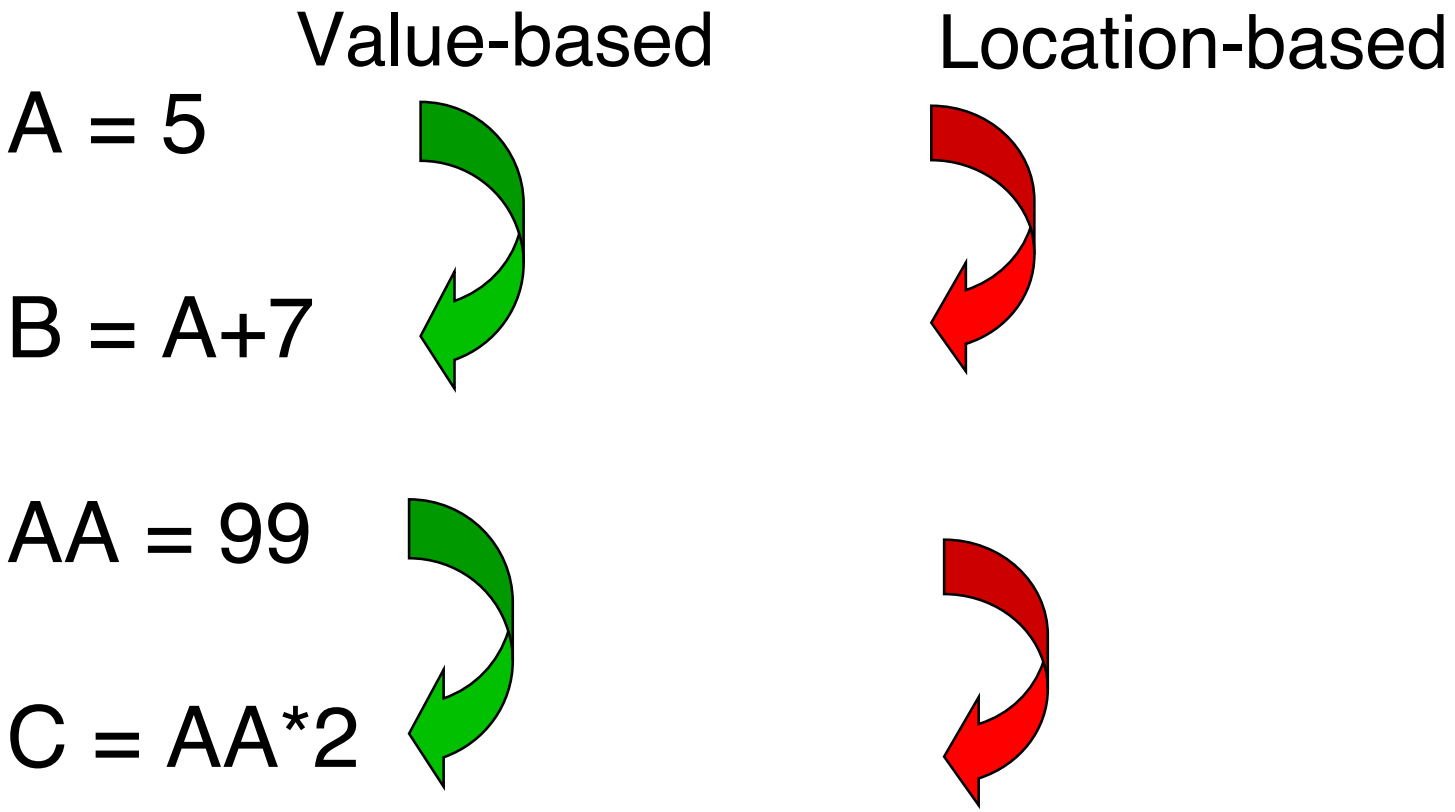
Location- vs. Value-Based

- Consider



By introducing new variables, dependencies can be removed

- Consider



Notation for Statements inside Loops

- If S is a statement inside a loop, then S(K) means the instance of the statement when the index value is K.

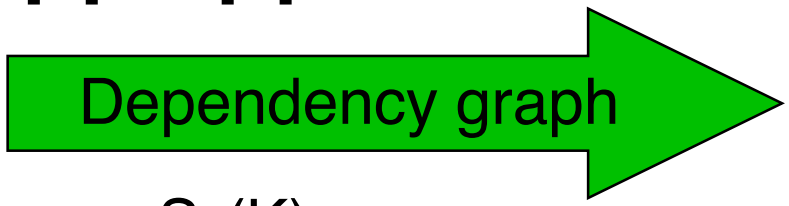
- Similarly, if there S is inside nested loops, then S(I, J, K, …) means the instance of S where the indices have values I, J, K, …

Loops add to the Challenge

- Consider

  do K= 1 to 10

  $S_1(K)$ A[K] = B[K]



- Conclude: All instances $S_1(K)$ can be done **concurrently** (since no arrows). e.g. use **doall**

(graph has no arcs in this case)

$S_1(1)$

$S_1(2)$

$S_1(3)$

.

.

.

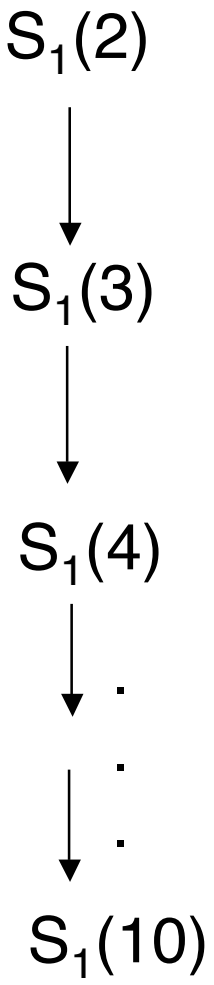
$S_1(10)$

Loops add to the Challenge

- Consider

  do K= 2 to 10

  $S_1(K)$ A[K] = A[K-1]



- Conclude: All instances $S_1(K)$ must be done in sequence.

  Note: The RHS could have been f(A[K-1]) for some function f.

Unroll Loop to Visualize

- do K= 2 to 10
  A[K] = A[K-1]

do K= 1 to 10
  A[K] = B[K]

Unrolled:

- A[2] = A[1]
  A[3] = A[2]
  A[4] = A[3]
  . . .

vs.

A[1] = B[1]
A[2] = B[2]
A[3] = B[3]

Loop-Carried Dependency

- do K= 2 to 10

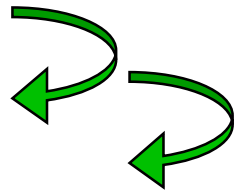
  A[K] = A[K-1]

Above, the loop introduces a dependency.
This is called a "**Loop-Carried Dependency**"

Loop Independent Dependency

- Dependencies that exist only **within** the loop body, and not across iterations, these dependencies are also called “Loop Independent Dependencies”.

```
doall I = 1, 100
        A(I) = C*A(I)
        B(I) = A(I)
        D(I) = X*B(I)
end
```



doall-ready

- If only loop-independent dependencies exist, the loop can be converted to a **doall** without changing its meaning.

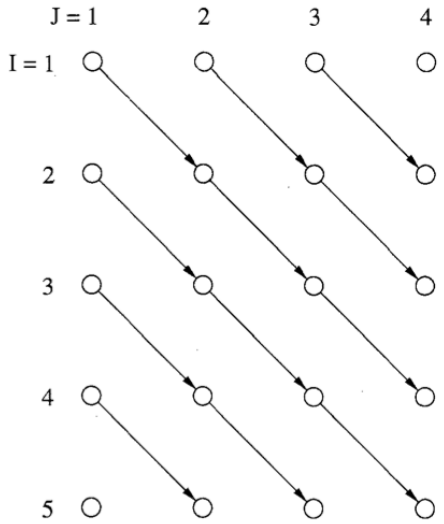
Iteration Space

- For a given (possibly nested) loop, the **iteration space** is the set of all possible tuples of loop index values, showing whether there are any dependencies between statements in the loop execution.

Iteration Space Example
(Wolfe and Banerjee)

```
      do I = 1, 5
          do J = 1, 4
S1:           X(I+1, J+1) = X(I, J) + Y(I, J)
          enddo
      enddo
```

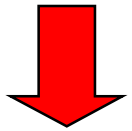
The data dependence relation $S_1 \, \delta \, S_1$ holds; in fact $S_1[I', J'] \, \delta \, S_1[I'+1, J'+1]$ for $1 \leqslant I' \leqslant 4$, $1 \leqslant J' \leqslant 3$. The iteration space with the particular arrows drawn is shown in Fig. 2.



“Forward” Dependencies

- Consider

“old” (previously-computed)



```
for K= 1 to 9
        A[K] = A[K+1]
```

A[1] = A[2]
A[2] = A[3]
A[3] = A[4]
. . .

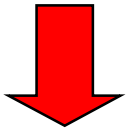
“Forward” Offsets

- Consider

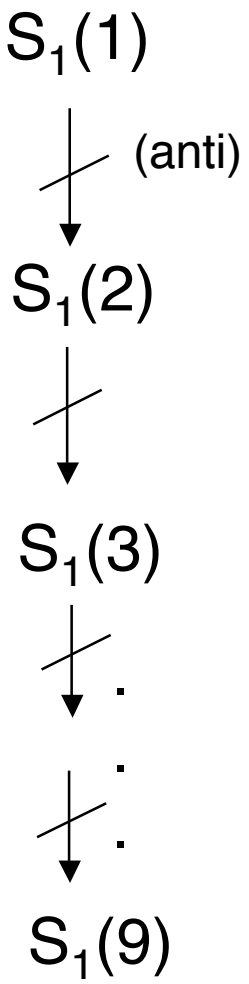
  for K= 1 to 9

  $S_1(K)$ A[K] = A[K+1]

“old” values

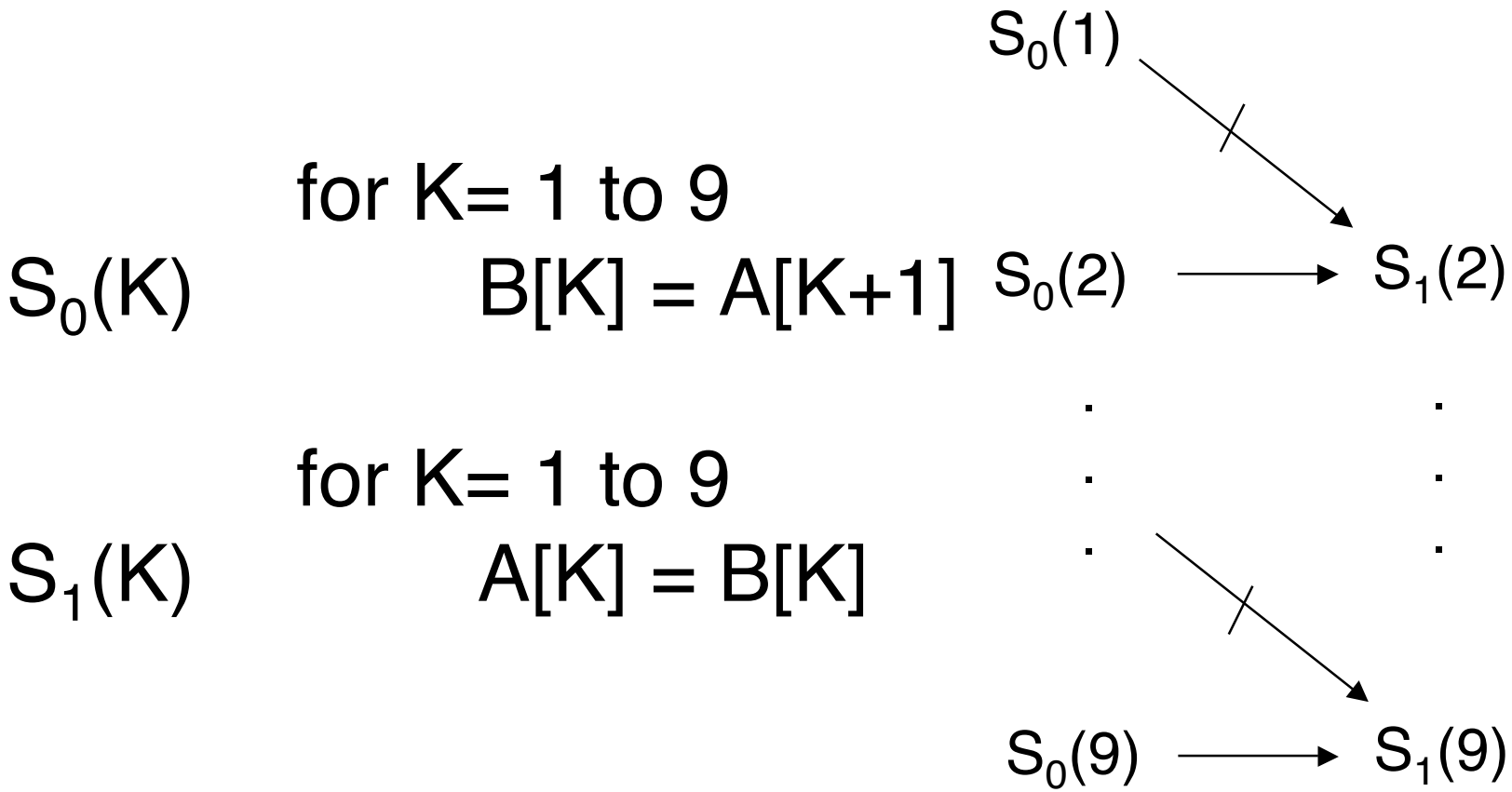


Dependency graph



- Conclude: Assuming **location-based**, rather than value-based, all instances $S_1(K)$ must be done in sequence (if location-based assumption used)

We could *Transform* the Previous Example by **buffering the old values** in new array B, if this is allowed.



Transformation reduces sequence constraints

for K= 1 to 9
S_0(K) B[K] = A[K+1]

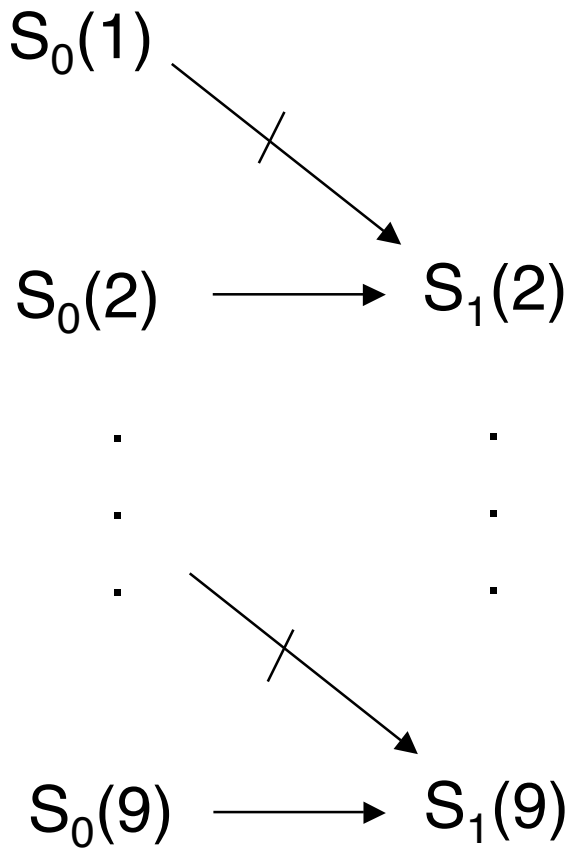
for K= 1 to 9
S_1(K) A[K] = B[K]

Then use doall or

Fortran 90 style:

B(1 : 9) = A(2 : 10)

A(1 : 9) = B(1 : 9)



Changing Evaluation Order

- We could also get the same effect by changing the order of iteration:

```
for K= 1 to 9
        A[K] = A[K+1]
```

becomes

```
for K= 9 to 1 by -1
        A[K] = A[K+1]
```

Parallel Execution of Loops Strategy

- Try to issue different instances of a loop body to separate processing elements.
- Generally loops occur nested; try to find an appropriate **nesting level** where different instances of the loop can be issued in parallel.

Parallelization vs. Vectorization Distinction

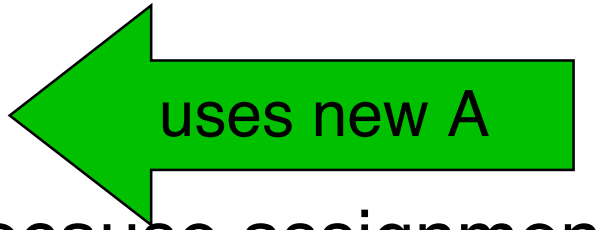
- **Parallelizing** concentrates on **outer** loops (coarser grain).
- **Vectorizing** concentrates on **inner** loop (fine grain).
- Vector machines:
  - Exploit parallel operations (+, -, *, /) on *vector* elements
  - Typically done with *vector registers*

Example of Loop Vectorization

- do K = 1 to N
  A[K] = B[K] + C[K]
  D[K] = A[K]*5

Vectorizes to (using Fortran 90 notation):

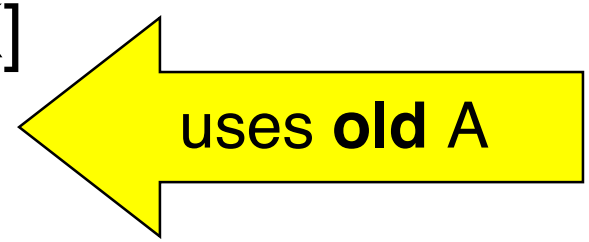
A(1:N) = B(1:N) + C(1:N)
D(1:N) = A(1:N)*5



No buffering is needed because assignment is done “all at once”.

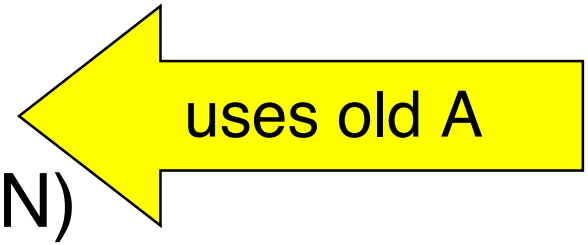
Example of Loop Vectorization with “forward” offsets

- do K = 1 to N
  A[K] = B[K] + C[K]
  D[K] = A[K+1]*5



Vectorizes to (using Fortran 90 notation):

-
  D(1:N) = A(2:N+1)*5
  A(1:N) = B(1:N) + C(1:N)



Note that the order has changed.

Dependence Distance

- Notation (where S_0 and S_1 are statements)

$$S_0(K) \ \delta^f_{(1)} \ S_1(K)$$

- This essentially says:
  - The K^{th} iteration of S_0 must be done before the $K + 1^{th}$ iteration of S_1.

Dependence Distance

- Similarly:

$$S_0(K)\ \delta^f_{(-1)}\ S_1(K)$$

- This essentially says:
  - The K^{th} iteration of S_0 must be done before the $K\text{-}1^{th}$ iteration of S_1.

Dependence Distance

- Similarly:

$$S_0(K) \ \delta^f_{(0)} \ S_1(K)$$

- This essentially says:
  - The K^{th} iteration of S_0 must be done in the **same** iteration of S_1.

Example of $\delta^f_{(1)}$

do K = 2 to N

$S_0(K)$ A[K] = B[K-1]

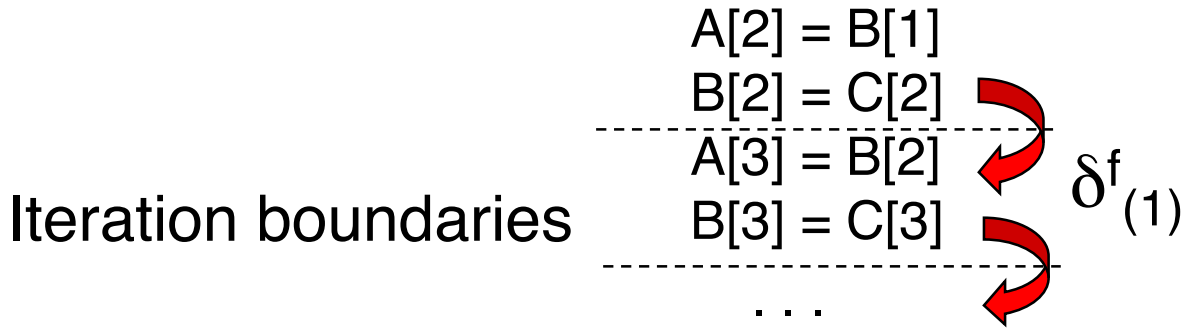
$S_1(K)$ B[K] = C[K]

$\delta^f_{(1)}$

Example of $\delta^f_{(1)}$

- do K = 2 to N
  S_0(K) A[K] = B[K-1]
  S_1(K) B[K] = C[K]

- Unrolled:



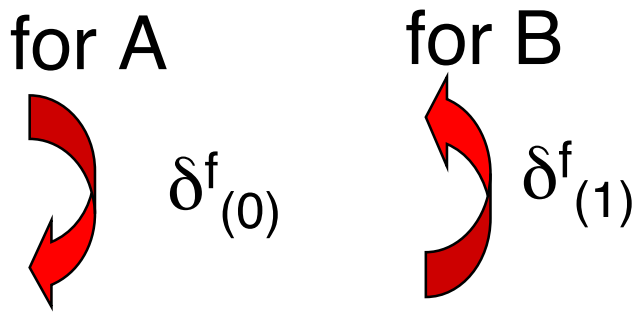
Per-Array Distances

- In general, there may be a different set of dependence distances **for each array**:

do K = 2 to N

$S_0(K)$ A[K] = B[K-1]

$S_1(K)$ B[K] = A[K]

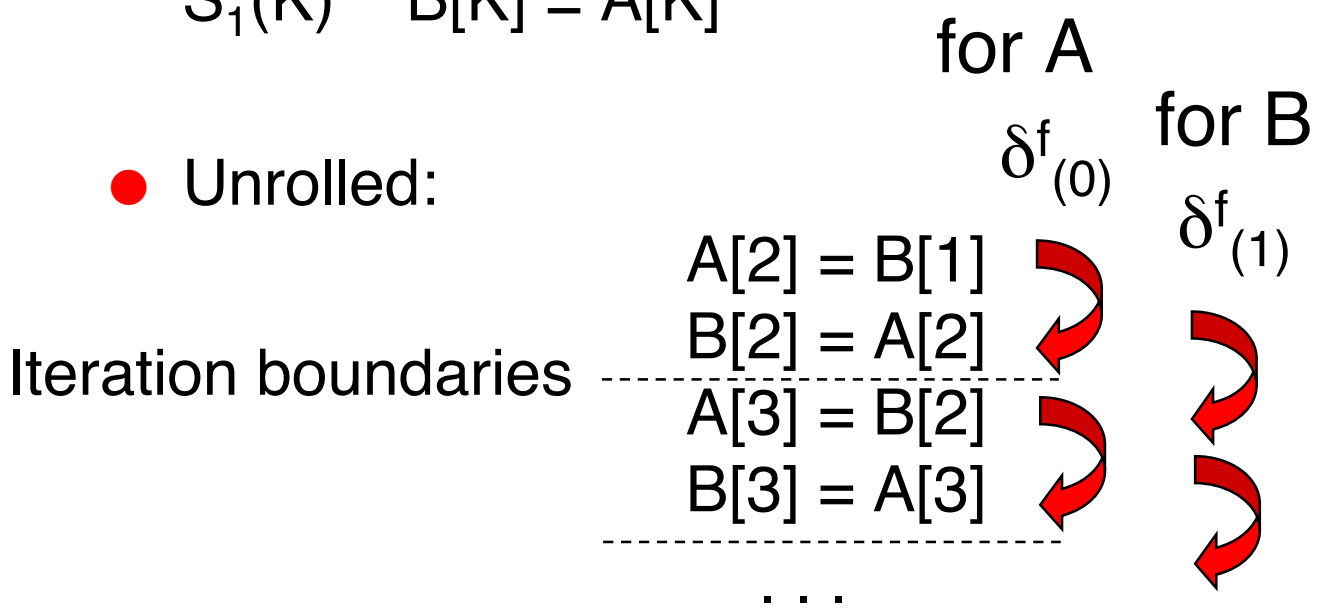


- The one for B says "the i^{th} instance of S_1 must be done before the $i+1^{th}$ iteration of S_0".
- The one for A says "the i^{th} instance of S_0 must be done before the i^{th} iteration of S_1".

Unrolling

- do K = 2 to N
  S_0(K) A[K] = B[K-1]
  S_1(K) B[K] = A[K]

- Unrolled:

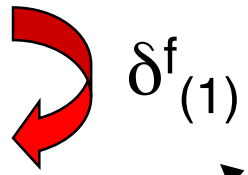


In general, both array indices and **Stride Direction** must be taken into account in determining Dependence Distance

- do K = 2 to N-1

S_0 A[K] = B[K]

S_1 C[K] = A[K-1]



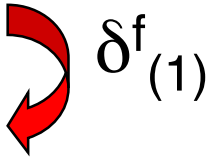
is similar to

negative stride

- do K = N-1 to 2 by -1

S_0 A[K] = B[K]

S_1 C[K] = A[K+1]



in that C[K] gets the *new* value, not the old.

same dependence distance:

The i^{th} iteration of S_0 must be done before the i +1th iteration of S_1.

Direction Vectors

- Less precise than Dependence Distances, but frequently used:
  - $\delta_{(<)}$ used in place of $\delta_{(n)}$ where $n > 0$
    (< to suggest **before (*not* that n < 0)**)
  - $\delta_{(=)}$ used in place of $\delta_{(0)}$
  - $\delta_{(>)}$ used in place of $\delta_{(n)}$ where $n < 0$

Direction Vectors

- *An* advantage of using general > rather than specific n is that n might not be fixed, as in:

```
do K = 2 to 10
        A[2*K] = B[K]+1
        C[K] = A[K]
```

- Here the dependence distance *increases* with K, rather than being uniform.

Examples

- do K = 2 to N

S_0 A[K] = B[K]

S_1 C[K] = A[K-1]

$\delta^f_{(1)}$ becomes $\delta^f_{(<)}$

do K = 1 to N-1

S_0 A[K] = B[K]

S_1 C[K] = A[K+1]

$\delta^a_{(1)}$ becomes $\delta^a_{(<)}$

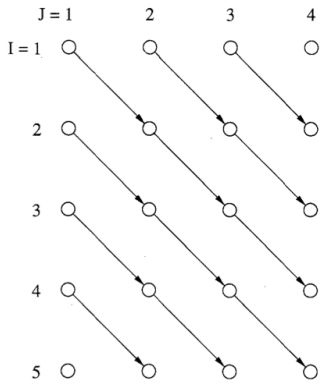
Nested Loops

- For nested loops, a **multi-dimensional** direction vector is needed

```
          do I = 1, 5
              do J = 1, 4
S1:               X(I+1, J+1) = X(I, J) + Y(I, J)
              enddo
          enddo
```

$S_1\ \delta_{(<,<)}\ S_2 \qquad S_1$

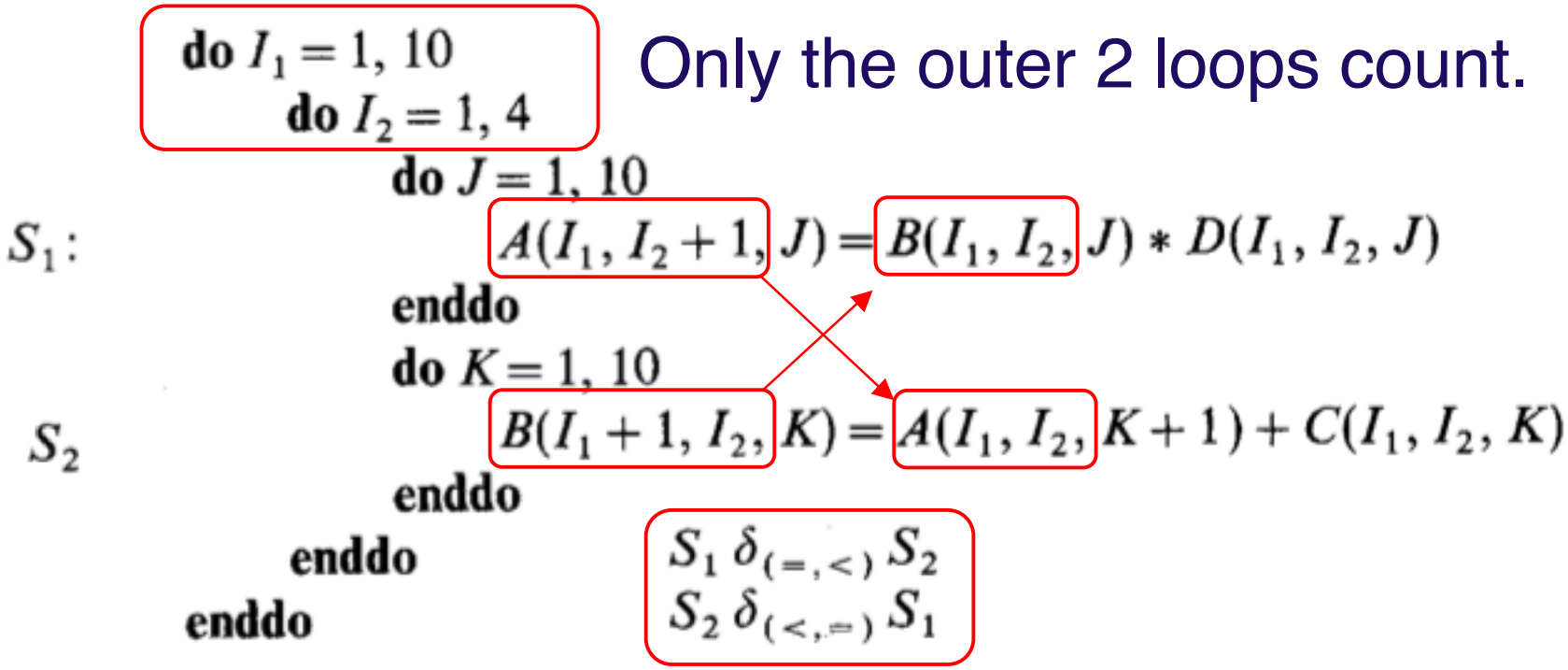
$S_1[1, 1]\ \delta\ S_2[2, 2]$



Direction Vectors for Sequential Loops inside a Nest

The sequential aspect is not given a direction.

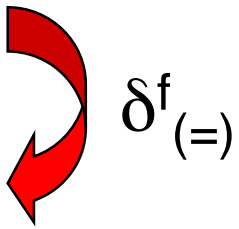
Only the outer 2 loops count.



doall Example (not loop-carried)

- Original loop

```
do K = 1 to N
        A[K] = C[K]
        B[K] = A[K]
```



is parallelized to

```
doall K = 1 to N
        A[K] = C[K]
        B[K] = A[K]
```

Loops that "Carry" Dependence

- As we saw, loops having *only* $\delta^{f}_{(=)}$ are parallelizable using doall.

- Loops with $\delta^{f}_{(<)}$ or $\delta^{f}_{(>)}$ carry dependences that constrains parallel execution.

Nested Loops

- For nested loops, a *vector* of dependences is used, e.g. $\delta^{f}_{(=,<)}$ or $\delta^{a}_{(=,=)}$ with one component per loop nest.

- When loops are nested, the ***outermost*** loop with a $\delta^{f}_{(<)}$ or $\delta^{f}_{(>)}$ is said to ***carry*** the dependence.

Parallelization ("Concurrentization" -- Wolfe)

- If a loop has only = dependence directions, then the iterations of it can be done **concurrently**.
- If loops *outer* to that loop have only < dependence directions, then those iterations are executed sequentially.
- Loops *inner* to that loop can be executed sequentially, or analyzed further.

Examples Using Direction Vectors

Direction Vector	**Outer Loop Iterations**	**Inner Loop Iterations**
(<, =)	Sequential	Parallel
(=, <)	Parallel	Sequential
(=, =)	Parallel	Parallel
(<, <)	Sequential	Sequential

Depth of Dependence

- The outermost nest level having a < direction is called the depth of dependence.
- That is the first level at which serial execution is required.

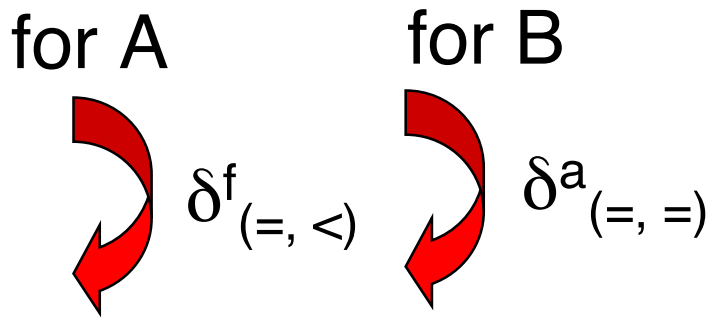
Also the "depth of a data dependence" is defined as the nest level of the outermost loop which must be executed serially to allow the dependence condition to be satisfied.[6,11] The depth of a data dependence is nothing more than the nest level of the outermost forwards ("<") direction in the direction vector for that dependence, and can thus also be derived from the direction vector of a dependence arc.

Examples of Dependence Depth

Direction Vector	Dependence Depth
(<, …)	1
(=, <, …)	2
(=, =, <, …)	3

Nested Loop Example
(=, <) depth = 2

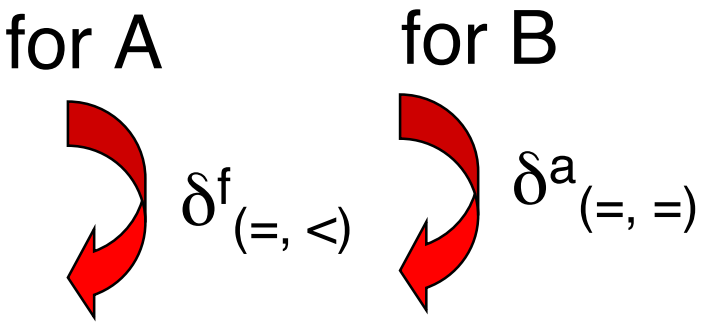
- do K = 2 to N
  do J = 2 to N
  A[K, J] = B[K, J]
  B[K, J] = A[K, J-1]



- The **inner** loop carries the dependence for A; no loop carries the dependence for B.
- Therefore the **outer** can be parallelized using doall.
- The depth of dependence is 2.

Nested Loop Example (=, <), continued

- do K = 2 to N
  do J = 2 to N
  A[K, J] = B[K, J]
  B[K, J] = A[K, J-1]



becomes

- doall K = 2 to N
  do J = 2 to N
  A[K, J] = B[K, J]
  B[K, J] = A[K, J-1]

Another (=, …) Example

```
      doall I = 2, N
            do J = 2, N
S1:         A(I, J) = (A(I, J − 1) + A(I, J + 1))/2
            enddo
      enddoall
```

Example

- Parallelizable?

```
do K = 2 to N
    do J = 2 to N
            A[K, J] = C[K, J]
            B[K, J] = A[K-1, J]
```

Example

- Parallelizable?:

```
do K = 2 to N
     do J = 2 to N
          A[K, J] = C[K, J]
          B[K, J] = A[K-1, J]
```

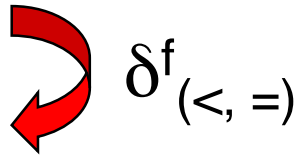
for A
$\delta^{f}_{(<, =)}$

- The outer loop carries the dependency.
- The depth is 1.
- The inner loop can be parallelized.

Example

- do K = 2 to N
  do J = 2 to N
  A[K, J] = C[K, J]
  B[K, J] = A[K-1, J]

for A



- Parallel:
  do K = 2 to N
  doall J = 2 to N
  A[K, J] = C[K, J]
  B[K, J] = A[K-1, J]

Change of Statement Order in Vectorization

```
      do I = 2, N
S1:       A(I) = B(I) + C(I)
S2:       D(I) = A(I+1) + 1
S3:       C(I) = D(I)
      enddo
```

$S_1\ \delta_{(=)}\ S_3$

$S_2\ \delta_{(=)}\ S_3$

$S_2\ \bar{\delta}_{(<)}\ S_1$

Code reordered for vector statements based on dependencies.

```
S2:  D(2:N) = A(3:N+1) + 1
S1:  A(2:N) = B(2:N) + C(2:N)
S3:  C(2:N) = D(2:N)
```

Loop Interchanging

- Two perfectly nested loops can be interchanged if there is no data dependence vector (<, >).

Loop Interchanging

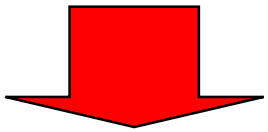
- do K = 1 to N
  do J = 1 to N
  S_1 A[K, J] = A[K, J-1] + A[K, J+1]

- Dependencies $S_1\ \delta^f_{(=,<)}\ S_1$ and $S_1\ \delta^a_{(=,<)}\ S_1$ imply **inner** loop **cannot** be vectorized.

- But no dependencies of form $\delta^f_{(<,>)}$ implies loops can be **interchanged**.

Loop Interchanging

- do K = 1 to N
  do J = 1 to N
  A[K, J] = A[K, J-1] + A[K, J+1]



- do J = 1 to N
  do K = 1 to N
  A[K, J] = A[K, J-1] + A[K, J+1]

- Now have $\delta^{f}_{(<, =)}$ so vectorizable.

Aside: Effect of a Dependence **Cycle** on Vectorization

When inspecting the data dependence graph of a loop, certain dependences can be ignored for vectorization. Any dependence relation that is satisfied by an outer serial loop need not be considered, as shown in the

```
      do I = 2, N - 1
         do J = 2, N - 1
S1:         T = A(I - 1, J) + A(I + 1, J)
S2:         A(I, J) = T
         enddo
      enddo
```

$S_1\ \delta_{(=,=)}\ S_2$ from T

$S_1\ \bar{\delta}_{(<,=)}\ S_2$ from A

$S_2\ \delta_{(<,=)}\ S_1$ from A

Even though there is a dependence cycle, the inner *do J* loop can be vectorized. The $S_2\ \delta_{(<,=)}\ S_1$ and $S_1\ \bar{\delta}_{(<,=)}\ S_2$ dependence relations are satisfied by the "<" direction on the *do I* loop, only the $S_1\ \delta_{(=,=)}\ S_2$ dependence needs to be considered when vectorizing the *do J* loop.

Aside: Self Anti- and Output- Dependencies Ignorable when Vectorizing

Some self-anti-dependence and self-output-dependence cycles can also be ignored when vectorizing loops. If fetches for right-hand side operands for iteration i are guaranteed to complete before the stores of any iteration $j(j>i)$, then any data anti-dependence from the right-hand side expression to the left-hand side variable can be ignored. Also, since most vector computers that allow indexed scatter operations will store the operands in index-set order, self-output dependences can be ignored.

```
      do I = 1, N
S1:       A(I) = A(I+1) - 1
S2:       B(IP(I)) = A(I)
      enddo
```

$S_1\ \bar{\delta}_{(<)}\ S_1$

$S_1\ \delta_{(=)}\ S_2$

$S_2\ \delta^{\circ}_{(<)}\ S_2$

However, the $S_1\ \bar{\delta}_{(<)}\ S_1$ dependence will usually be satisfied by the way code is generated (fetches for $A(I+1)$ will be performed before stores for $A(I)$). Also, for many machines, the $S_2\ \delta^{\circ}_{(<)}\ S_2$ dependence will be satisfied since the store for $S_2[i]$ will always be completed before the store for $S_2[j]$ for all $j>i$.

Aside: Reductions can cause Relaxing of Constraints

```
        do I = 1, N
            do J = 1, N                                   S1 δ(=,=) S2
S1:             A(I, J) = B(I, J) + C(I, J)
S2:             AMAX = MAX(AMAX, A(I, J))                 S2 δ(≤,*) S2
            enddo
        enddo
```

Even though the dependence relation $S_2\ \delta_{(<,>)}\ S_2$ holds, we will allow loop interchanging here because the dependence appears only in a reduction.

Larger offsets allow more concurrency

- Consider offset 2:

```
do K= 3 to 10
        A[K] = A[K-2]
```

- A[3] = A[1]
- A[4] = A[2]

} Concurrency within groups

- A[5] = A[3]

…

- Similarly, A[K] = A[K-d] will allow degree d concurrency.
- Can use **doacross,** not **doall**

“Strip-Mining”

- aka loop-blocking

```
for(i=0; i<N; ++i){
   ...
}
```

can be blocked with a block size B by replacing it with

```
for(j=0; j<N; j+=B)
  for(i=j; i<min(N, j+B); ++i){
     ....
  }
```

Loop-Tiling

- Extension of strip-mining to >1 dimensions

Original matrix multiplication:

```
DO I = 1, M
 DO K = 1, M
   DO J = 1, M
      Z(J, I) = Z(J, I) + X(K, I) * Y(J, K)
```

After loop tiling B*B:

```
DO K2 = 1, M, B
  DO J2 = 1, M, B
    DO I = 1, M
      DO K1 = K2, MIN(K2 + B - 1, M)
         DO J1 = J2, MIN(J2 + B - 1, M)
            Z(J1, I) = Z(J1, I) + X(K1, I) * Y(J1, K1)
```

2x2 tiled matrix-vector multiply

```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
  c[i] = 0;
  for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
      for (y = j; y < min(j + 2, n); y++) {
        c[x] = c[x] + a[x][y] * b[y];
      }
    }
  }
}
```

Disclaimer

We have only skimmed the surface of a very rich field that has been on-going since the mid 1960's.