TU Wien

Institute of Information Systems Engineering
Research Division of Compilers and Languages

Übersetzer für Parallele Systeme
Compilers for Parallel Systems

LVA 185.A64
SS 2016

Hans Moritsch

hans.moritsch@tuwien.ac.at

Vectorization

* Vector statements specify rectangular *array sections* using *triplet notation*
* Fetch-before-Store semantics
  * every input on the right hand side of an assignment is loaded from memory before any element of the result is stored
  * ⇔ all inputs mentioned on the right hand side refer to their values before the statement is executed

```
for i=1 to m step 2
   a[i,j]=b[i-1,m+1,-j]
endfor
```

⇔

```
a[i,1:m:2] = b[i-1,m:1:-2]
```

```
for i=1 to n
   a[i]=b[i]+1
endfor
```

⇔

```
a[1:n] = b[1:n] + 1
```

```
for i=1 to 64
   a[i+1]=a[i]+b[i]
endfor
```

☹

```
a[2:65] = a[1:64] + b[1:64]
```

Vectorization

- Loop with single statement that carries no dependence can be directly vectorized (by 2.8)

```
for i=1 to n
    x[i] = x[i]+c
endfor
```

→

```
x[1:n] = x[1:n] + c
```

correct ☺

- Loop with single statement that carries a true dependence cannot be directly vectorized

```
for i=2 to n
  S x[i] = x[i-1]+c
endfor
```

→

```
x[2:n] = x[1:n-1] + c
```

incorrect ☹

old values of x (prior to loop execution) are used

$S\ \delta_1\ S$

iteration i uses new value of x (from previous iteration i-1)

Vectorization of Single Statement Loops

Vector statement generation is valid if there is no dependence $S\ \delta\ S$.

Vectorization

- Can any statements in loops that carry (forward) dependences be directly vectorized?

```
for i=1 to n
 S1 a[i+1] = b[i] + c
 S2 d[i]   = a[i] + e
endfor
```

```
S1 a[2:n+1] = b[1:n] + c
S2 d[1:n]   = a[1:n] + e
```

correct ☺

S_2 *uses new values of a assigned by* S_1

$S_1 \, \delta_1^{\downarrow} \, S_2$

```
---------------
a[1] = b[0] + c
d[0] = a[0] + e
---------------
a[2] = b[1] + c
d[1] = a[1] + e
---------------
```

S_2 *uses new value of a assigned by* S_1

A dependence $S \, \delta \, S'$ is *backward* if S' *bef* S, otherwise it is *forward*.

Vectorization

- Can any statements in loops that carry (forward) dependences be directly vectorized?

```
for i=1 to n
 S1 a[i+1] = b[i] + c
 S2 d[i]   = a[i] + e
endfor
```

$S_1\ \delta_1^{\downarrow}\ S_2$

```
---------------
a[1] = b[0] + c
d[0] = a[0] + e
---------------
a[2] = b[1] + c
d[1] = a[1] + e
---------------
```

S_2 uses new value of a assigned by S_1

```
S1 a[2:n+1] = b[1:n] + c
S2 d[1:n]   = a[1:n] + e
```

correct ☺

S_2 uses new values of a assigned by S_1

Loop distribution

```
for i=1 to n
 S1 a[i+1] = b[i] + c
endfor
for i=1 to n
 S2 d[i]   = a[i] + e
endfor
```

$\neg(S_1\ \delta\ S_1)$, $\neg(S_2\ \delta\ S_2)$

$\Rightarrow$ can directly vectorize both loops

Vectorization

* Can also statements in loops that carry backward dependences be vectorized?

```
for i=1 to n
 S1 d[i]   = a[i] + e
 S2 a[i+1] = b[i] + c
endfor
```

if input values for an iteration (S_1) originate from a previous iteration (S_2), we may split the loop only if S_1 is before S_2

$S_2 \, \delta_1^{\uparrow} S_1$

```
---------------
d[1] = a[1] + e
a[2] = b[1] + c
---------------
d[2] = a[2] + e
a[3] = b[2] + c
---------------
```

S_1 uses new value of a assigned by S_2

Loop distribution ?

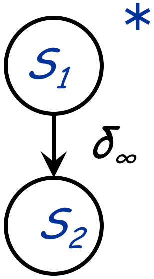
```
for i=1 to n
 S1 d[i]   = a[i] + e
endfor
for i=1 to n
 S2 a[i+1] = b[i] + c
endfor
```

incorrect ☹

S_1 could never use a value assigned by S_2

Vectorization
Two Statements

* Loop with two statements and loop independent dependence

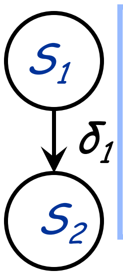



```
for i=1 to n
  S1 x[i] = …
  S2  …   = x[i]+c
endfor
```

```
S1 x[1:n] = …
S2  …      = x[1:n] + c
```

☺

* loop carried *forward* dependence

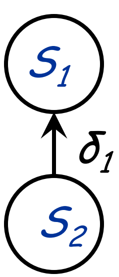



```
for i=2 to n
  S1  x[i] = …
  S2 …      = x[i-1]+c
endfor
```

```
S1 x[2:n] = …
S2   …      = x[1:n-1] + c
```

☺

* loop carried *backward* dependence



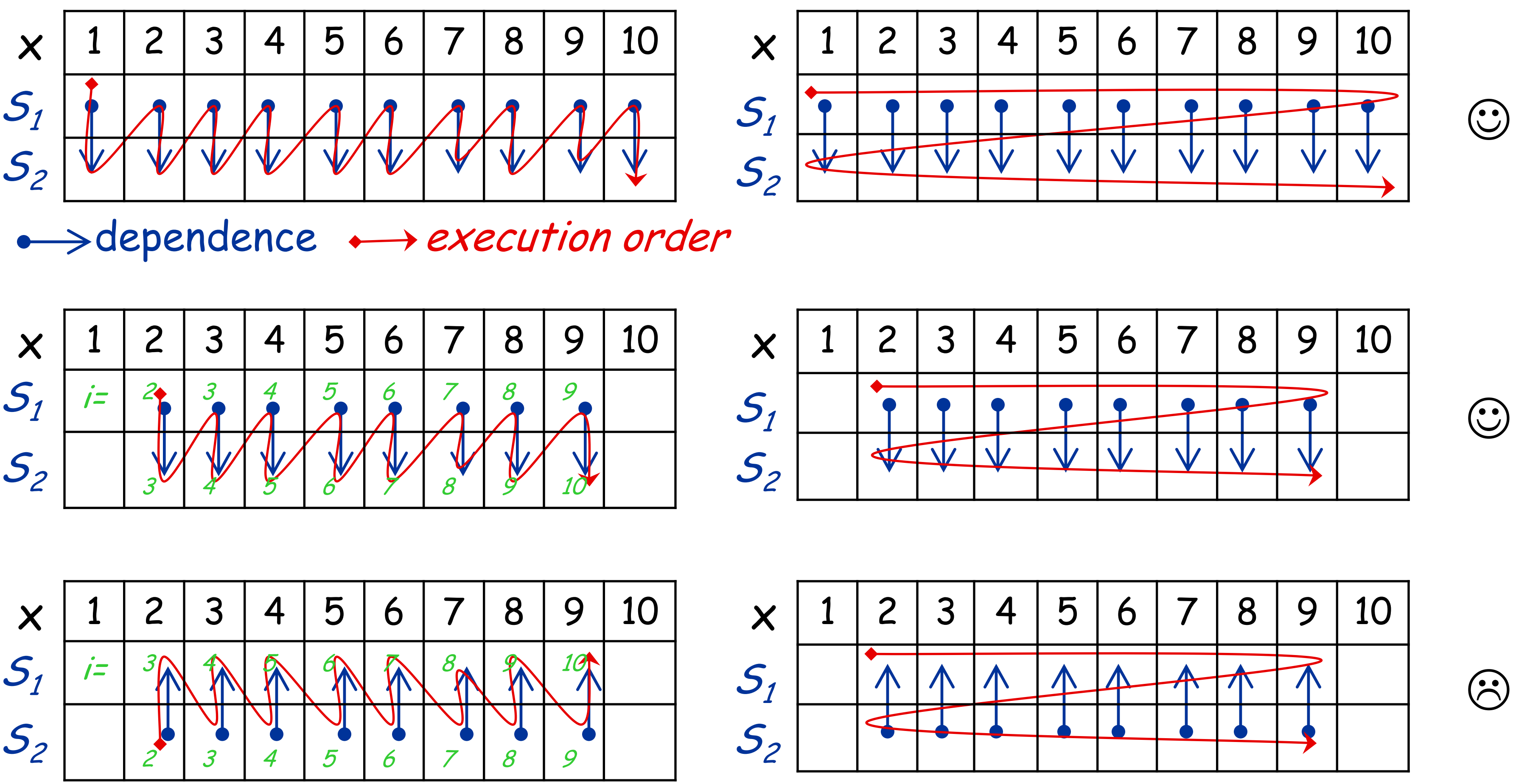

```
for i=2 to n
 S1    …  = x[i-1]+c
 S2 x[i] = …
endfor
```

```
S1        …   = x[1:n-1] + c
S2 x[2:n] = …
```

☹

Vectorization
Two Statements

original | vectorized



dependences are not preserved!

Statement Reordering

```
for i=1 to 100
 S1  a[i] = b[i] * 2
 S2  c[i] = a[i-1]-4
endfor
```

```
for i=1 to 100
   S2  c[i] = a[i-1]-4
   S1  a[i] = b[1] * 2
endfor
```

☺

```
for i=1 to 100
 S1  a[i] = b[i] * 2
 S2  c[i] = a[i]-4
endfor
```

```
for i=1 to 100
S2 c[i] = a[i]-4
S1 a[i] = b[i] * 2
endfor
```

☹

Statement Reordering

- Statement reordering, applied to a loop and statements S_1 and S_2, is valid iff there is no loop independent dependence from S_1 and S_2.
  - if there are only loop carried dependences between S_1 and S_2, all dependences are preserved

Vectorization

* Can also statements in loops that carry backward dependences be vectorized?

```
for i=1 to n
 S1  d[i]   = a[i] + e
 S2  a[i+1] = b[i] + c
endfor
```

$S_2\,\delta_1^{\uparrow}\,S_1$

```
---------------
d[1] = a[1] + e
a[2] = b[1] + c
---------------
d[2] = a[2] + e
a[3] = b[2] + c
---------------
```

S_1 uses new value of a assigned by S_2

```
S2  a[2:n+1] = b[1:n] + c
S1  d[1:n]   = a[1:n] + e
```

correct ☺

S_1 uses new values assigned by S_2

Statement reordering

```
for i=1 to n
S2  a[i+1] = b[i] + c
S1  d[i]   = a[i] + e
endfor
```

$S_2\,\delta_1^{\downarrow}\,S_1$

$\neg(S_1\,\delta_\infty\,S_2)$

$\Rightarrow$ interchanging S_1 and S_2 is possib (correct) by ...

Vectorization Examples

Statement reordering

```
for i=1 to 100
  S1 d[i] = a[i-1] * d[i]
  S2 a[i] = b[i] + c[i]
endfor
```

```
for i=1 to 100
  S2 a[i] = b[i] + c[i]
  S1 d[i] = a[i-1] * d[i]
endfor
```

Loop distribution

```
for i=1 to 100
  S2 a[i] = b[i] + c[i]
endfor
for i=1 to 100
  S1 d[i] = a[i-1] * d[i]
endfor
```

Vector statement generation

```
S2 a[1:100] = b[1:100] + c[1:100]
S1 d[1:100] = a[0:99] * d[1:100]
```

```
for i=1 to n
   for i=1 to n
      S c[i,j] = c[i-1,j] * d[i-1,j+1]
   endfor
endfor
```

Vector statement generation

```
S c[i,1:n] = c[i-1,1:n] – d[i-1,2:n-1]
```

Vectorization

- Problem: backward loop carried <u>and</u> loop independent dependence

```
for i=1 to n
 S1  b[i]   = a[i] + e
 S2  a[i+1] = b[i] + c
endfor
```

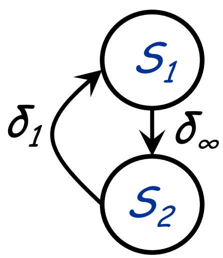
$S_2\ \delta_1^{\uparrow}\ S_1$ $\quad$ $S_1\ \delta_\infty\ S_2$

```
----------------
b[1] = a[1] + e
a[2] = b[1] + c
----------------
b[2] = a[2] + e
a[3] = b[2] + c
----------------
```

```
for i=1 to n
 S2  a[i+1] = b[i] + c
 S1  b[i]   = a[i] + e
endfor
```

☹

reordering S_1 and S_2 reverses
$S_1\ \delta_{\infty,true}\ S_2$ to $S_2\ \delta_{\infty,anti}\ S_1$



- Vectorization of a statement
  - distribute the loop around it
  - ensure all inputs to the distributed loop are computed before
  - precluded by dependence cycle

Loop Distribution Basic Version

```
for i=1 to n
   for j=1 to n
       S1 c[i,j] = 0.0
      for k=1 to n
      S2 c[i,j] = c[i,j] + a[i,k] * b[k,j]
      endfor
   endfor
endfor
```

```
for i=1 to n
   for j=1 to n
       S1 c[i,j] = 0.0
   endfor
endfor
for i=1 to n
   for j=1 to n
      for k=1 to n
      S2 c[i,j] = c[i,j] + a[i,k] * b[k,j]
      endfor
   endfor
endfor
```

- Single statement loops
- Each statement is executed for its whole execution index set, before any statement that textually follows

Loop Distribution

* A loop is *distributive* iff loop distribution (basic version) can be validly applied to it.
* Every backward dependence is loop-carried.

Distributive Loop

* A loop is distributive iff all its dependences are forward.

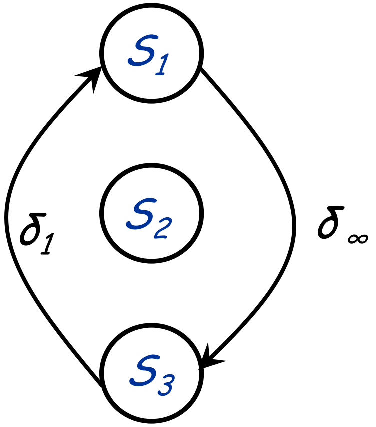
Distributive Loop / Reordering

* A loop can be made distributive by a sequence of valid statement reordering transformations iff its dependence graph is either acyclic or contains only elementary loops.

Loop Distribution Example

```
for i=1 to n
    S1 c[i]=a[i-2]*b[i]
    S2 d[i]=b[i]+b[i-1]/2
    S3 a[i]=c[i]+2
endfor
```

```
for i=1 to n
    S1 c[i]=a[i-2]*b[i]
    S3 a[i]=c[i]+2
endfor
S2 d[1:n]=b[1:n]+b[0:n-1]/2
```



* dependence cycle involving more than one statements
* which are not adjacent

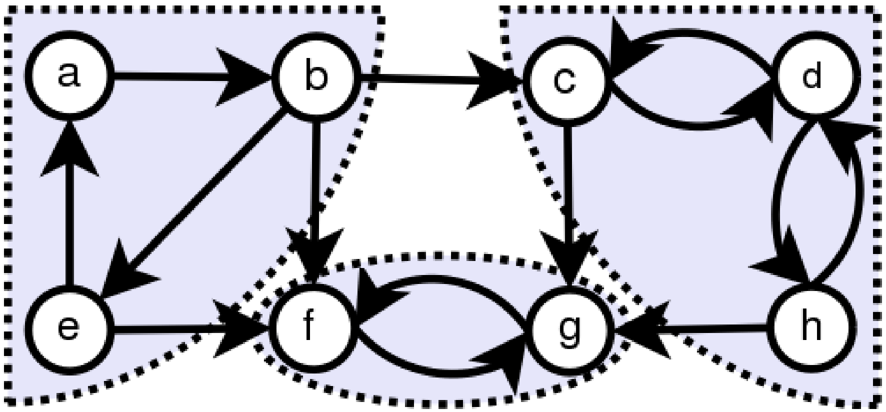
Loop Distribution General Version

```
for i=1 to n
   for j=1 to n
       S1 a[i,j] = …
       S2 a[i, j+1] = a[i,j] -1
   end for
   S3 b[i] = 2 * d[i] + 3
   S4 c[i] = b[i] -4
endfor
```

```
for i=1 to n
   for j=1 to n
       S1 a[i,j] = …
       S2 a[i, j+1] = a[i,j] -1
   end for
 end for
 for j=1 to n
    S3 b[i] = 2 * d[i] + 3
 end for
 for j=1 to n
    S4 c[i] = b[i] -4
endfor
```

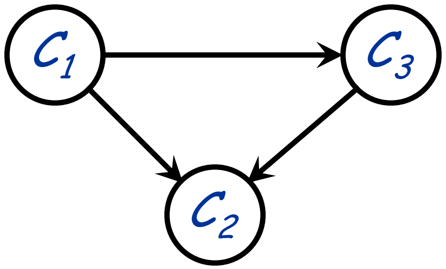
- Multiple statement loops
- **Blocks** of adjecent statements are executed for the whole execution index set

Strongly Connected Components of a Directed Graph



http://en.wikipedia.org/wiki/File:Scc.png

aka. π-blocks



acyclic condensation

compute SCCs e.g. with Tarjan's algorithm

Loop Vectorization Algorithm Simple Version

- (1) Find statements in dependence cycles
  - compute strongly connected components of dependence graph
- (2) Sort SCCs topologically
- (3) For every SCC (in topological order)
  - if SCC is cyclic
    - generate loop around the statements of the SCC
  - else (SCC is a single statement not involved in a cycle)
    - generate vector code for the statement

Loop Vectorization Algorithm Simple Version

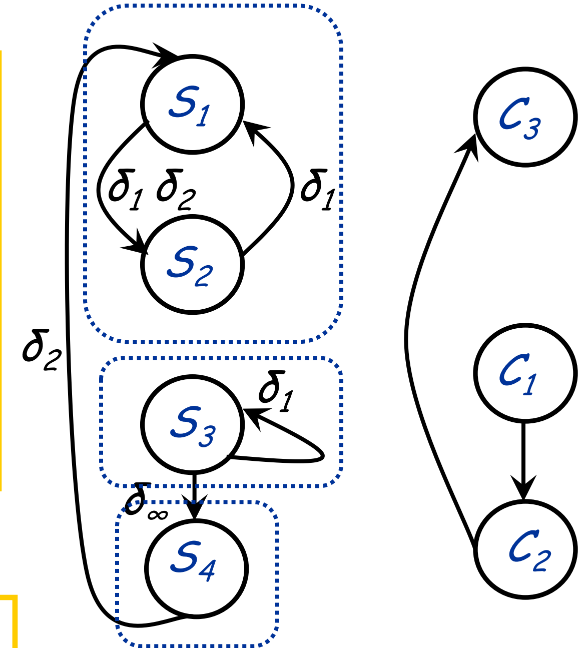
- (1) Find statements in dependence cycles
  - compute strongly connected components of dependence graph
- (2) Sort SCCs topologically
- (3) Reorder statements of loop body
  - according to topological order of SCCs
  - statements belonging to an SCC are adjacent
- (4) Apply loop distribution
- (5) Generate vector code for loops corresponding to a single statement SCCs (not involved in a cycle)

Vectorization Example

```
for i=1 to n
   for j=1 to m
      S1 c[i,j]=a[i,j]*b[i,j]
      S2 a[i+1,j+1]=c[i,j-2]/2+c[i-1,j]*3
      S3 d[i,j]=d[i-1,j-1]+1
      S4 b[i,j+4]=d[i,j]-1
   endfor
endfor
```

Statement reordering

```
for i=1 to n
   for j=1 to m
      S3 d[i,j]=d[i-1,j-1]+1
      S4 b[i,j+4]=d[i,j]-1
      S1 c[i,j]=a[i,j]*b[i,j]
      S2 a[i+1,j+1]=c[i,j-2]/2+c[i-1,j]*3
   endfor
endfor
```



SCCs

Vectorization Example

Loop distribution

```
for i=1 to n
   for j=1 to m
      S3 d[i,j]=d[i-1,j-1]+1
   endfor
endfor
for i=1 to n
   for j=1 to m
      S4 b[i,j+4]=d[i,j]-1
   endfor
endfor
for i=1 to n
   for j=1 to m
      S1 c[i,j]=a[i,j]*b[i,j]
      S2 a[i+1,j+1]=c[i,j-2]/2+ …
   endfor
endfor
```

Vector statement generation

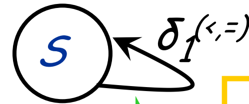
```
for i=1 to n
   for j=1 to m
      S3 …
   endfor
endfor

S4 b[1:n,5:m+4]=d[1:n,1:m]-1

for i=1 to n
   for j=1 to m
      S1 …
      S2 …
   endfor
endfor
```

Limitations of Basic Vectorization

```
for i=1 to n
   for j=1 to m
   S a[i+1,j] = a[i,j]+b
   endfor
endfor
```

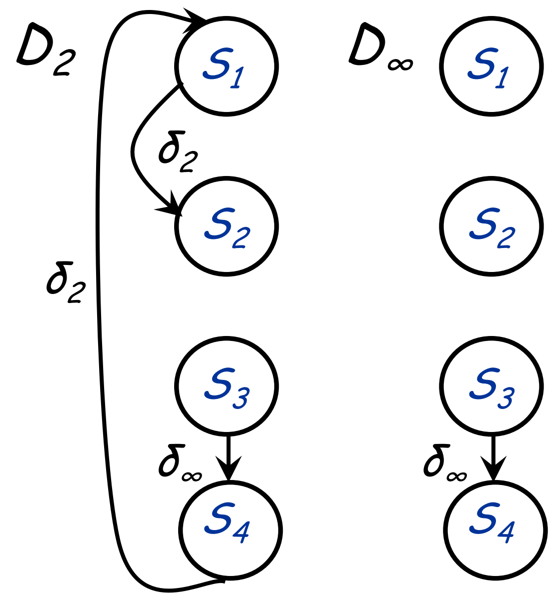



```
for i=1 to n
S a[i+1:1:m] = a[i,1:m]+b
endfor
```

- Run outer loop sequentially to preserve the dependence
- ⇒ Vectorize inner level if there is a recurrence at an outer level
- Recursive Approach
  - try to generate vector code at the outermost loop level
  - if impossible, run the outer loop sequentially
  - try again one level deeper, ignoring dependences carried by the outer loop

Concurrency in Loops

- How can we utilize parallelism at specific levels within a nested loop ?
- *levels(e)* of an edge
  - *minlevel(e), maxlevel(e)*
- Layered dependence graph D_k
  - only level ≥ k dependences
  - $k \in N \cup \{\infty\}$
- *levels(π)* of a path
  - levels maintained along a path
  - k is a level of a path, if
    - $k \in$ *levels(e)* for at least one edge in π
    - $k \le$ *maxlevel(e)* for all edges of π
  - *minlevel(π), maxlevel(π)*
  - *maxlevel(π)* = min{*maxlevel*(e): e in π}



Concurrency in Loops

- Consider statement *S* in a loop at level n, and level c at this or outer loop level: **c ≤ n**
  - **iff** there exists a cyclic path π inclosing *S* and c ϵ *levels* (π)
    - *S* is ***serial*** at level c
  - **else**
    - *S* is ***concurrent*** at level c

```
Lc for ic=1 to …
      …
      Ln for in=1 to …
         S …
         …
      endfor
endfor
```

- deeper than loop level: **c > n**
  - *S* is ***scalar*** at level c

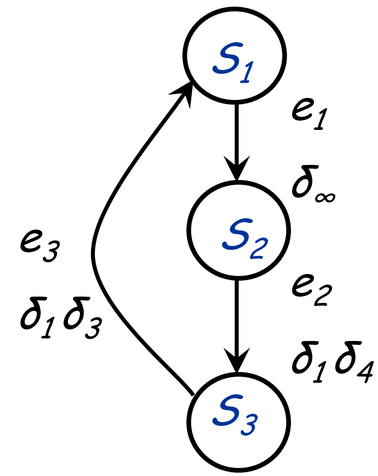
```
Ln for in=1 to …
      …
      S …
      Lc for ic=1 to …
         …
      endfor
endfor
```

Concurrent Instances

- If *S* is concurrent at level c, the instances of *S* in L_c can be executed concurrently.

Concurrency in Loops
Example

```
L1 for i=1 to n
     L2 for j=1 to m
          L3 for k=1 to n
                 S1
               L4 for l=1 to m
                     S2
                     …
                     S3
                 endfor
            endfor
       endfor
  endfor
```

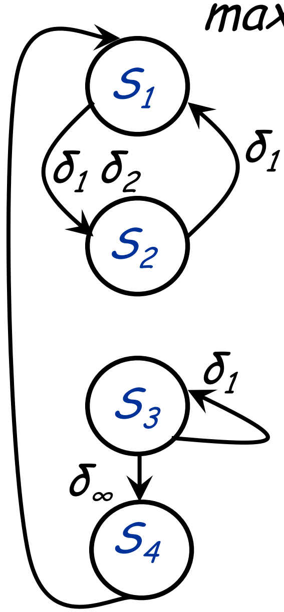


- cyclic path $\pi = (S1, S2, S3, S1)$
- $levels(e1) = \{\underline{\infty}\}$
- $levels(e2) = \{1,\underline{4}\}$
- $levels(e1) = \{1,\underline{3}\}$
- $levels(\pi) = \{1,3\}$
- $maxlevel(\pi) = \min\{\infty,4,3\} = 3$

- *S1:* loop level n=3
  - serial at level $1 \leq 3$
  - concurrent at level $2 \leq 3$
  - serial at level $3 \leq 3$
  - scalar at level $4 > 3$
- *S2, S3*: loop level n=4
  - serial at level $1 \leq 4$
  - concurrent at level $2 \leq 4$
  - serial at level $3 \leq 4$
  - concurrent at level $4 \leq 4$

Concurrency in Loops

* Vectorize a statement *S* at loop level n in as many inner loops than possible
  * *maxcycle*(*S*): greatest (deepest) level such that *S* is in a dependence cycle at that level
* = 0, if *S* is not in a dependence cycle at all
  * *vectorization index:* n - *maxcycle* (*S*)



maxcycle	*vect. index*
1	1
1	1
1	1
0	2

```
for i=1 to n
  S3 d[i,1:m]=d[i-1,0:m-1]+1
endfor

S4 b[1:n,5:m+4]=d[1:n,1:m]-1

for i=1 to n
  S1 c[i,1:m]=a[i,1:m]*b[i,1:m]
  S2 a[i+1,2:m+1]=c[i,-1:m-2]/2+ …
endfor
```

Partial Vectorization

- Run loop outer loops sequentially
  - including level *maxcycle* (*S*)
- Vectorize inner loops

Partial Vectorization $m = maxcycle(S)$

```
L1 for i1=lb1 to ub1
     L2 for i2=lb2 to ub2
          …
           Lm for im=lbm to ubm
                    S(i1,…,im,lbm+1:ubm+1, … ,lbn:ubn)
              endfor
          endfor
      endfor
```

Allen-Kennedy Vector Code Generation

procedure vectorcode(R, c)

R … code region to vectorize: set of assigment statements

c … loop level

- **construct** layered dependence graph D_c of R
- **construct** SCCs of D_c
- **for** every SCC in topological order
  - **if** SCC is cyclic

    R' … statements of the SCC: generate loop around R'
    - **generate**("`for` $\mathtt{i}_c$ `= lb`$_c$ `to ub`$_c$");
    - **vectorcode**(R', $c + 1$);
    - **generate**("`endfor`");
  - **else** (SCC is a statement S not involved in a cycle)

    generate vector code for S
    - **generate**("`S(i`$_1$`,…,i`$_{c-1}$`,lb`$_c$`:ub`$_c$`, … ,lb`$_n$`:ub`$_n$");

Allen-Kennedy Vector Code Generation Example

- **vectorcode**($\{S_1,S_2,S_3,S_4\}$, **1**)
  - SCCs = ($\{S_3\}$, $\{S_4\}$, $\{S_1,S_2\}$)
  - $C_1=\{S_3\}$ is **cyclic** $\Rightarrow$
  - **generate**("`for i_c = lb_c to ub_c`");
  - **vectorcode**($\{S_3\}$, **2**)
    - SCCs = ($\{S_3\}$)
    - $\{S_3\}$ is acyclic $\Rightarrow$ **generate**("S_3`(i,1:m)`")
  - **generate**("`end for`");
  - $C_2=\{S_4\}$ is acyclic $\Rightarrow$ **generate**("S_4`(1:n,1:m)`")
  - $C_3=\{S_1,S_2\}$ is **cyclic** $\Rightarrow$
  - **generate**("`for i_c = lb_c to ub_c`");
  - **vectorcode**($\{S_1,S_2\}$, **2**)
    - SCCs = ($\{S_1\}$, $\{S_2\}$)
    - $\{S_1\}$ is acyclic $\Rightarrow$ **generate**("S_1`(i,1:m)`")
    - $\{S_2\}$ is acyclic $\Rightarrow$ **generate**("S_2`(i,1:m)`")
  - **generate**("`endfor`");

complexity $O(d*(n+e))$
depth of loop nest, nr. of nodes
and edges in dependence graph