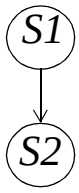
Chapter 9:

DEPENDENCE-DRIVEN LOOP MANIPULATION

9.1 DEPENDENCES

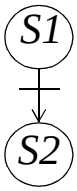
Flow Dependence (True Dependence)

```
S1    X=A+B
S2    C=X+1
```



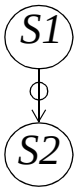
Anti Dependence

```
S1    A=X+B
S2    X=C+D
```

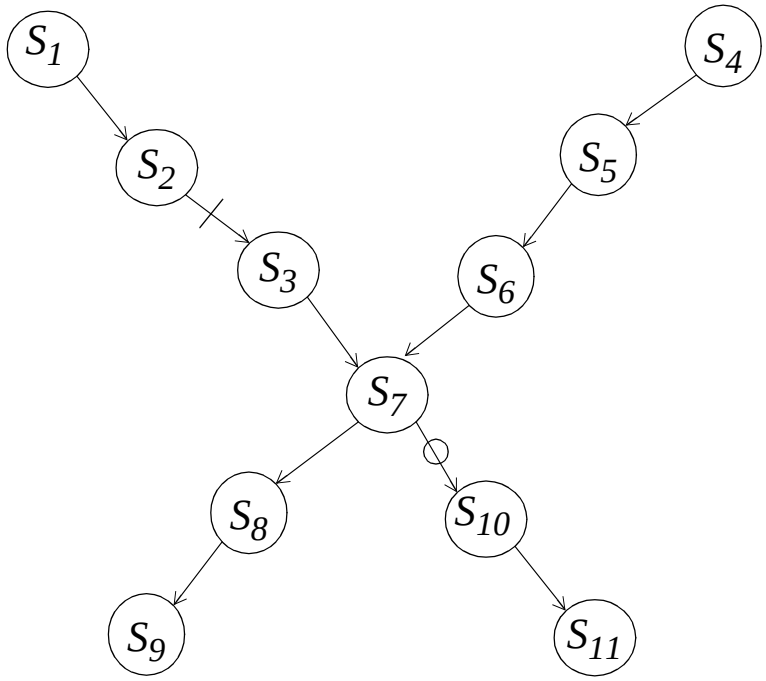


Output Dependence

```
S1    X=A+B
      . . .
S2    X=C+D
```



9.2 DEPENDENCE AND PARALLELIZATION (SPREADING)



S_1; S_2; S_3 can execute in parallel with S_4; S_5; S_6

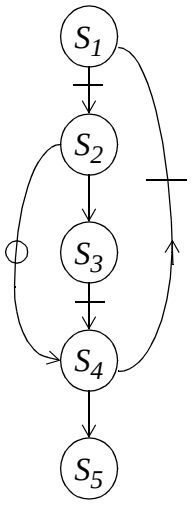
S_8; S_9 " " " " " S_{10}; S_{11}

```
C$OMP     SECTIONS

C$OMP     SECTION A
          S1
          S2
          S3
C$OMP     SECTION B
          S4
          S5
          S6

C$OMP     END SECTIONS

          S7

C$OMP     SECTIONS
C$OMP     SECTION
          S8
          S9
C$OMP     SECTION
          S10
          S11
C$OMP     END SECTIONS
```

9.3 RENAMING

(To remove memory-related dependences)

```
S1    A=X+B
S2    X=Y+1
S3    C=X+B
S4    X=Z+B
S5    D=X+1
```

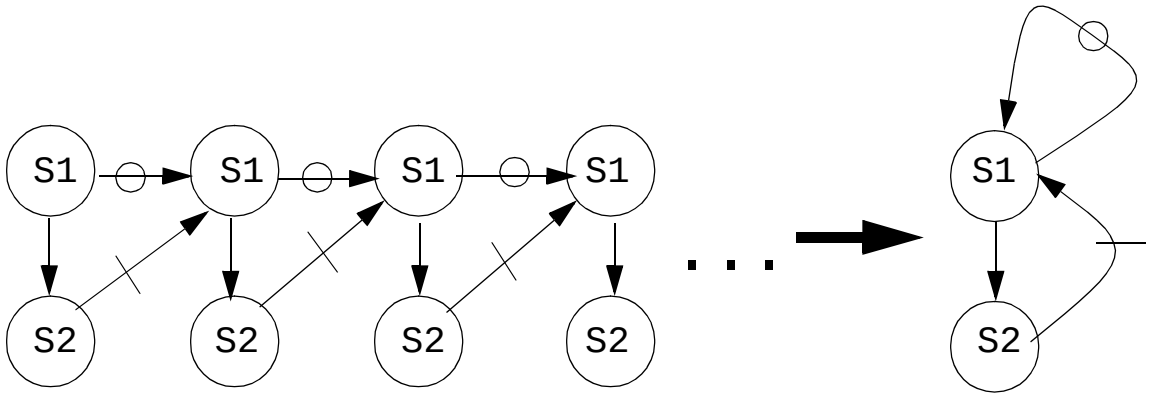


Use renaming.

```
S1    A=X+B
S2    X1=Y+1
S3    C=X1+B
S4    X2=Z+B
S5    D=X2+1
```

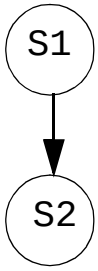
9.4 DEPENDENCES IN LOOPS

```
      DO I=1,N
S1        A=B(I)+1
S2        C(I)=A+2
      END DO
```

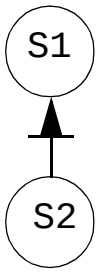


9.5 DEPENDENCES IN LOOPS (Cont.)

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

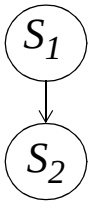



```
     DO I=1,N
S1         X(I)=B(I)+1
S1         A(I)=X(I+1)+1
     END DO
```



9.6 DEPENDENCE ANALYSIS

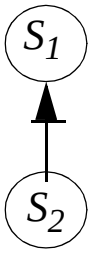
```
     DO I=1,N
S1         X(F(I)) = B(I)+1
S2         A(I) = X(G(I))+2
     END DO
```

We say that 

IFF $\exists$ $I_1 \leq I_2$

$\ni$ F (I_1)=G(I_2)

[ALSO $I_1, I_2 \varepsilon [1,N]$]

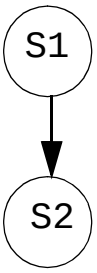
We say that 

IFF $\exists$ $I_1 < I_2$

$\ni$ F (I_2)=G(I_1)

9.7 LOOP PARALLELIZATION AND VECTORIZATION

- A loop whose dependence graph is cycle-free can be parallelized or vectorized.
  e.g.




```
DO I=1,N
     X(I)=B(I)+1
     A(I)=X(I)+1
END DO
```

```
X(1:N)=B(1:N)+1
A(1:N)=X(1:N)+1
```

```
PARALLEL DO I=1,N
     X(I)=B(I)+1
     A(I)=X(I)+1
END PARALLEL DO
```

- The reason is that if there are no cycles in the dependence graph, then there will be no races in the parallel loop.

9.8 ALGORITHM REPLACEMENT

- Some program patterns occur frequently in programs. They can be replaced with a parallel algorithm.
  e.g.

```
DO I=1,N
     A(I)=A(I-1)+B(I)
END DO
```

```
A(1:N)=REC1N(B(1:N),A(0),N)
```

```
X=A(1)
DO I=2,N
     IF(X.GT.A(I))X=A(I)
END DO
```

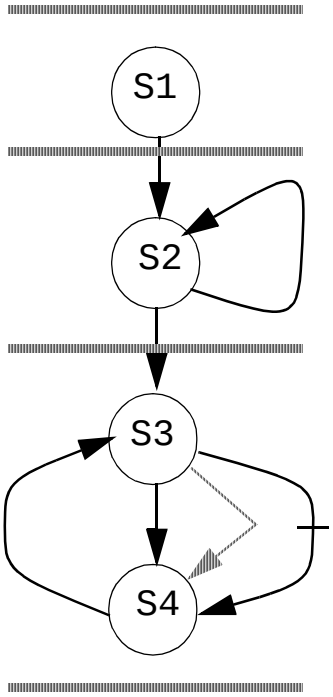
```
X=MIN(A(1:N))
```

9.9 LOOP DISTRIBUTION

- To insulate these patterns, we can decompose loops into several loops, one for each strongly-connected component (π-block)in the dependence graph.

```
DO I=1,N
S1:       A(I)-B(I)+C(I)
S2:       D(I)=D(I-1)+A(I)
S3:       IF(X.GT.A(I))THEN
S4             X=A(I)
          ENDIF
     END DO
```

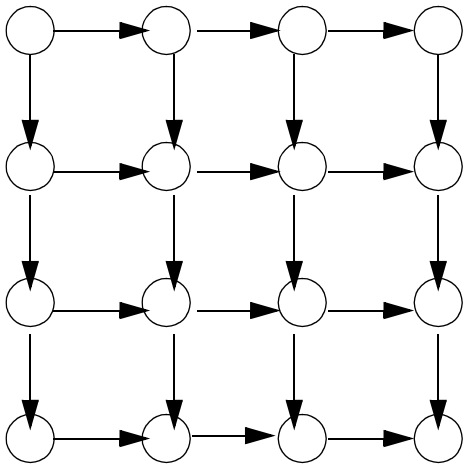
```
     DO I=1,N
          A(I)=B(I)+C(I)
     END DO
     DO I=1,N
          D(I)=D(I-1)+A(I)
     END DO
     DO I=1,N
          IF (X.GT.A(I) THEN
               X=A(I)
          END IF
     END DO
```



9.10 LOOP INTERCHANGING

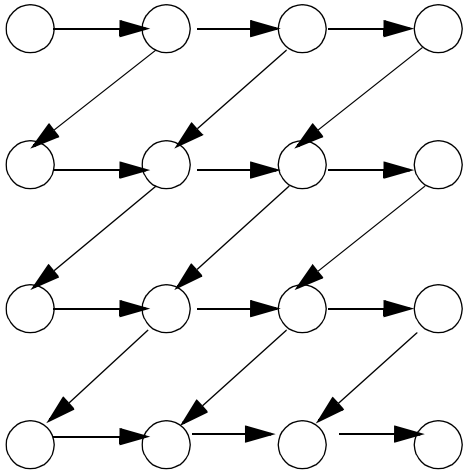
- The dependence information detremines whether or not the loop headers can be interchanged.
- For example, the following loop headers can be interchanged

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



- However, the headers in the following loop cannot be interchanged

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

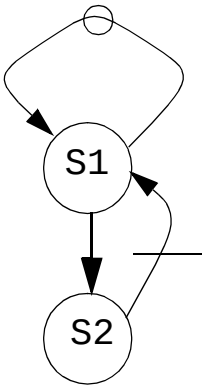


9.11 DEPENDENCE REMOVAL

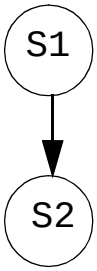
- Some cycles in the dependence graph can be eliminated by using elementary transformations.

Scalar Expansion:

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

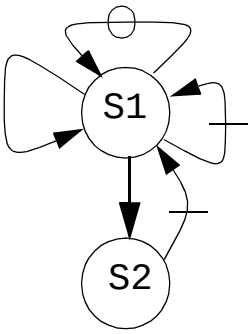



```
     DO   I=1,N
S1:       A1(I)=B(I)+1
S2:       C(I)=A1(I)+D(I)
     END DO
     A=A1(N)
```



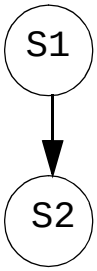
9.12 Induction variable recognition

```
     DO I=1,N
S1:       J=J+2
S2:       X(I)=X(I)+J
     END DO
```




```
     DO I=1,N
S1:       J1=J+2*I
S2:       X(I)=X(I)+J1
     END DO
```

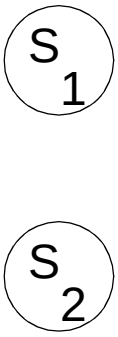
```
     DO I=1,N
S1:       J1(I)=J+2*I
S2:       X(I)=X(I)+J1(I)
     END DO
```



9.13 More about the DO to PARALLEL DO transformation

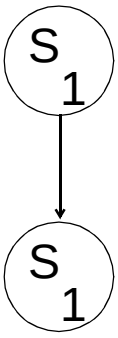
- When the dependence graph inside a DO loop has no cross-iteration dependences, it can be transformed into a PARALLEL DO.

Example 1:




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

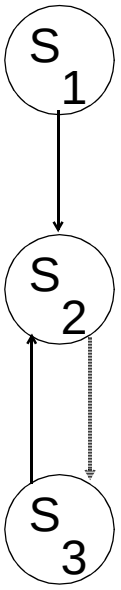
Example 2:

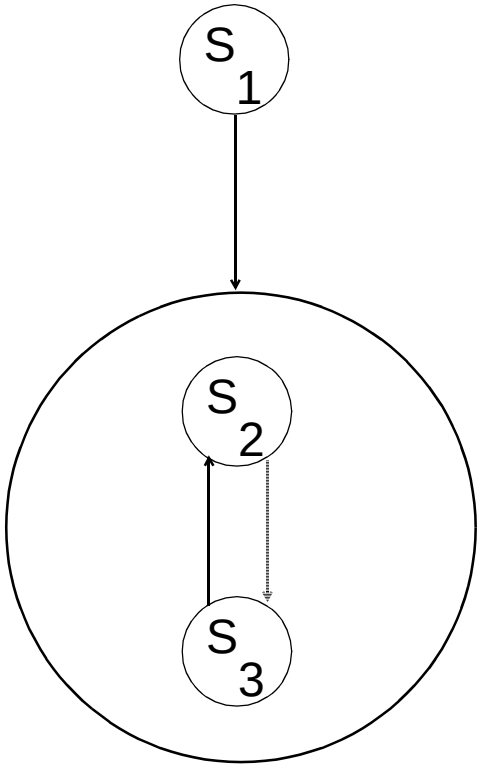



```
do i=1,n
S1 :  a(i) = b(i) + c(i)
S2 :  d(i) = x(i) + 1
end do
```

Example 3:

```
do i=1,n
S  :  b(i) = a(i)
 1
S  :  do while b(i)**2-a(i).gt.epsilon
 2
S  :        b(i)=(b(i)+a(i)/b(i))/2.0
 3
      end do while
end do
```

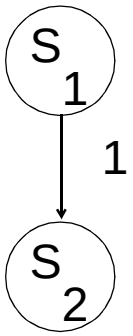




- When there are cross iteration dependences, but no cycles, do loops can be *aligned* to be transformed into DOALLs

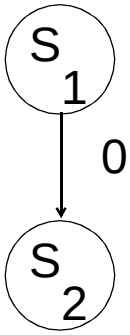
  Example 1:

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



↓

```
do i=0,n
S  :  if i>0 then a(i) = b(i) + 1
 1
S  :  if i<n then c(i+1) = a(i)**2
 2
end  do
```



- Sometimes we have to replicate to achieve alignment

Example 2:

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

↓

```
do i=1,n
      a(i) = b(i) + c(i)
      a1(i) = b(i) + c(i)
      d(i) = a1(i) + a(i-1)
end do
```

↓

```
do i=0,n
      if i>0 then a(i) =b(i) + c(i)
      if i<n then a1(i+1)=b(i+1)+c(i+1)
                  d(i+1)=a1(i+1)+a(i)
end do
```

- Need for replication could propagate.

  Example 3:

```
do i=1,n
      c(i) = 2 * f(i)
      a(i) = c(i) + c(i-1)
      d(i) = a(i) + a(i-1)
end do
```

↓

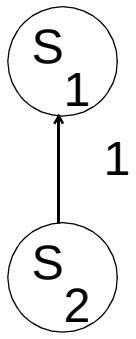
```
do i=1,n
      c(i) = 2 * f(i)
      c1(i) = 2 * f(i)
      c2(i) = 2 * f(i)
      a(i) = c(i) + c1(i-1)
      a1(i) = c1(i) + c2(i-1)
      d(i) = a(i) + a1(i-1)
end do
```

- The problem of finding the minimum amount of code replication sufficient to align a loop is NP-hard in the size of the input loop (Allen et al 1987)

- To do alignment, we may need to do topological sort of the statements according to the partial order given by the dependence graph.

  Example 4:

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



- Performing alignment without sorting first will clearly be incorrect in this case. However, by "peeling off" the first iteration of S_1 and the last iteration of S_2, we would achieve the desired result.:

```
S1 : a(1) = b(1) + c(0)
do i=1,n-1
S2 : c(i) = d(i)
S1 : a(i+1) = b(i+1) + c(i)
end do
S2 : c(n) = d(n)
```

- Another method for eliminating cross-iteration dependences is to perform loop distribution.

Example:

```
do i=1,n
      a(i) = b(i) + 1
      c(i) = a(i-1) + 2
end do
```

↓

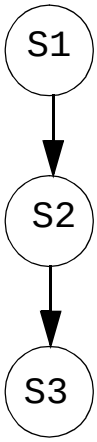
```
do i=1,n
      a(i) = b(i) + 1
end do
do i=1,n
      c(i) = a(i-1) + 2
end do
```

9.14 Vectorization

1. Loops wioth a cycle-freer dependence graphs can be easily vectorized.

   (a) In the simplest case, each statement in the body can be translated into a vector statement:

```
do i=1,n
S  :  a(i) = b(i) + c(i)
  1
S  :  d(i) = a(i-1)+x(i)
  2
S  :  e(i) = d(i)+t
  3
end do
```

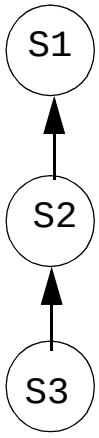


↓

```
a(1:n)=b(1:n)+c(1:n)
d(1:n)=a(0:n-1)+x(1:n)
e(1:n)=d(1:n)+t
```

(b) In more complex cases, topological sorting is needed:

```
do i=1,n
S  :  a(i) = b(i-1) + c(i)
  1
S  :  b(i) = d(i)+x(i)
  2
S  :  d(i+2) = w(i)+t
  3
end do
```

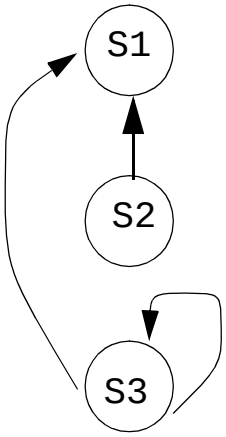


↓

```
d(2:n+1)=w(1:n)+t
b(1:n)=d(1:n)+x(1:n)
a(1:n)=b(0:n-1)+c(1:n)
```

2. And when cycles are present, in the dependence graph, a loop can sometimes be vectorizable.

```
do i=1,n
S  :  a(i) = b(i-1) + d(i-1)
  1
S  :  b(i) = t+x(i)
  2
S  :  d(i) = d(i-1)+y(i)
  3
end do
```



↓

```
d(1:n)=lr1(d(0),y(1:n))
b(1:n)=x(1:n)+t
a(1:n)=b(0:n-1)+d(0:n-1)
```

9.15 Loop Coalescing for DOALL loops

- A perfectly nested DOALL loop such as

```
doall i=1,n1
      doall j=1,n2
            doall k=1,n3
                  ...
            end doall
      end doall
end doall
```

could be trivially transformed into a singly-nested loop with a tuple of variables as index:

```
doall (i,j,k) = (1..n1).c.(1..n2).c.(1..n3)
      ...
end doall
```

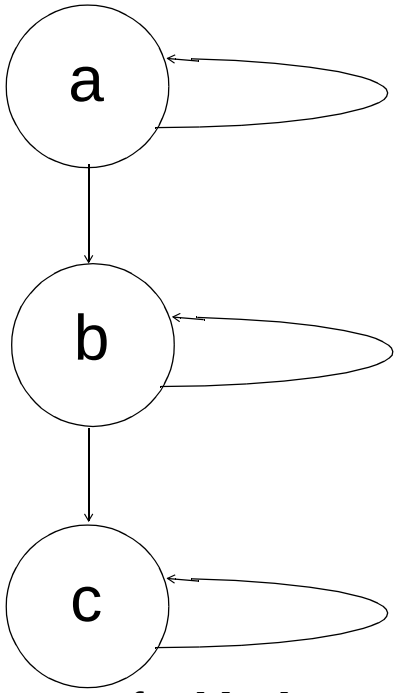
This coalescing transformation is convenient for scheduling and could reduce the overhead involved in starting DOALL loops.

9.16 Cyclic Dependences -- DOPIPE

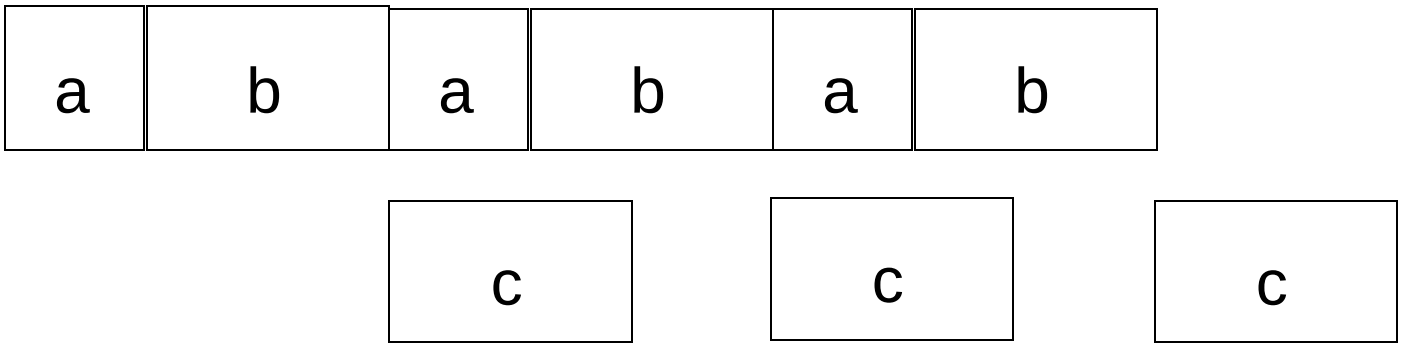
- Assume a loop with two or more dependence cycles (strongly connected components or π-blocks)
- The first approach developed for concurrentization of do loops is illustrated below:

```
do i=1,n
      a(i) = b(i) + a(i-1)
      c(i) = a(i) + c(i-1)
end do
            ↓
cobegin
      do i=1,n
            a(i) = b(i) + a(i-1)
            V(σ)
      end do
//
      do i=1,n
            P(σ)
            c(i) = a(i) + c(i-1)
      end do
coend
```

i.e. to take a loop with two or more π-blocks such as:

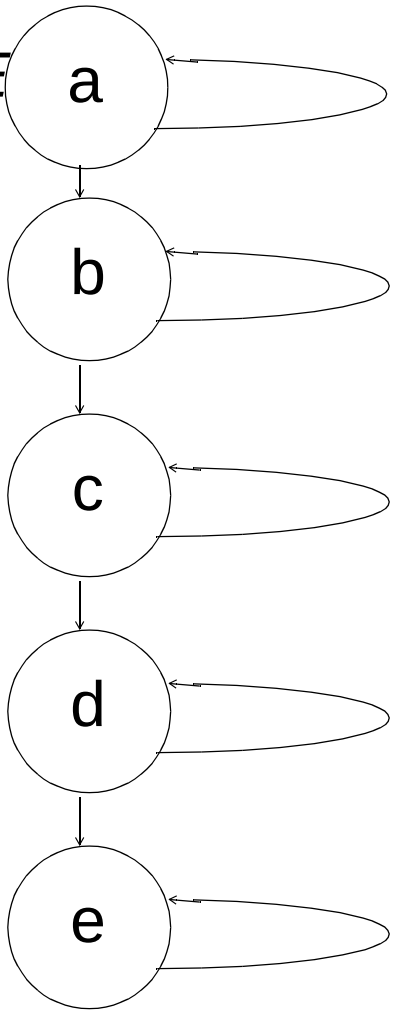


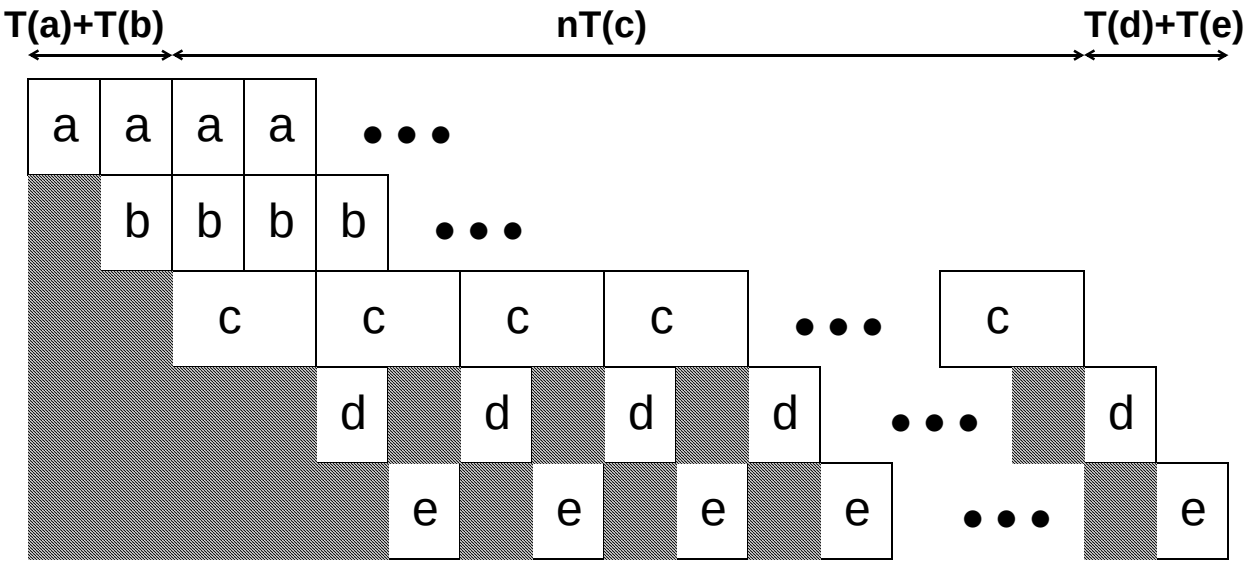
and execute collections of π-blocks on separate processors in a pipelined fashion:



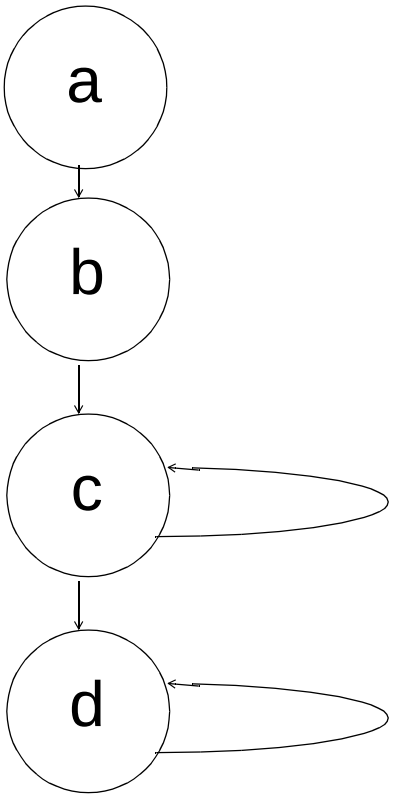
9.16.1 Execution time of DOPIPE

- Assume the dependence graph shown to the right. Assume also that T(c)=max(T(a),T(b),T(c),T(d),T(e)) Then the execution time of the DOPIPE on 4 processors is T(a)+T(b)+nT(c)+T(d)+T(e)





9.16.2 DOPIPE and Loop Distribution

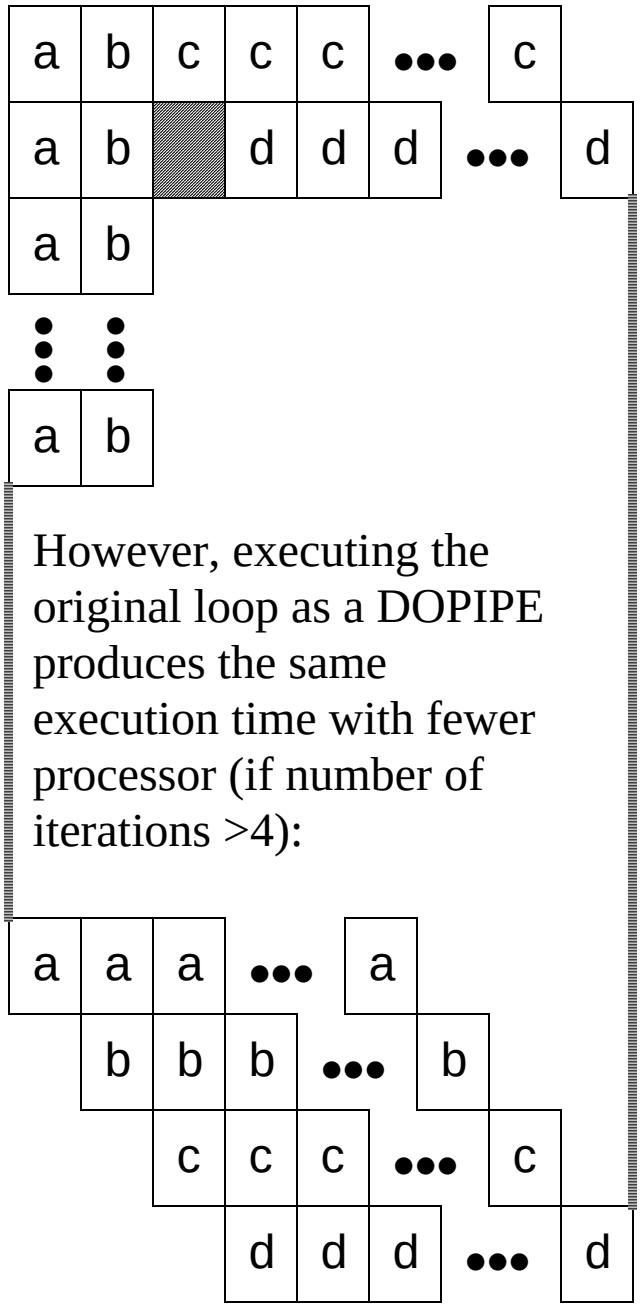


Assume a loop with the dependence graph shown on the right

The loop could be distributed to produce:

```
do i=1,n
      a
      b
end do
do i=1,n
      c
      d
end do
```

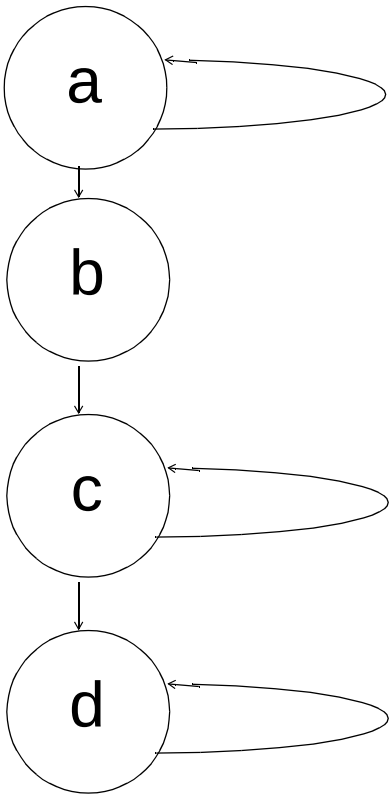
The first loop could be transformed into a DOALL, and the second into a DOPIPE. The resulting time lines would be:


a b c c c ●●● c
a b d d d ●●● d
a b
a b
However, executing the original loop as a DOPIPE produces the same execution time with fewer processor (if number of iterations >4):
a a a ●●● a
b b b ●●● b
c c c ●●● c
d d d ●●● d

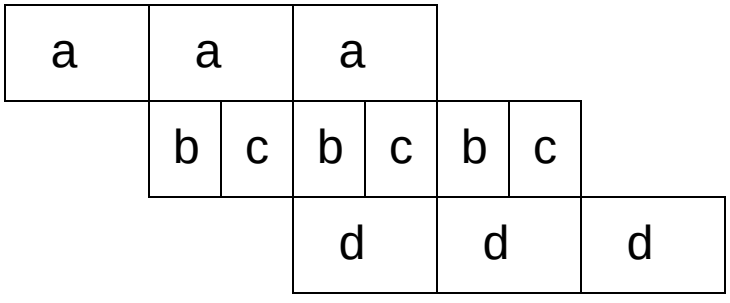
9.16.3 Problems with DOPIPE

1. Processor allocation is fixed at compile-time, i.e. loops are compiled for a fixed number of processors.

   Example 1: A loop with the dependence graph shown to the right, could be compiled for three processors as:

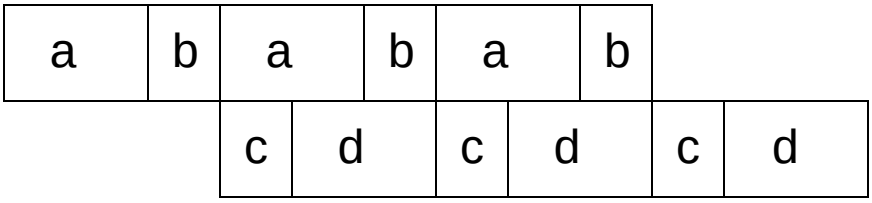



```
cobegin
        do i=1,n
              a
        end do
//
        do i=1,n
              b
              c
        end do
//
        do i=1,n
              d
        end do
coend
```

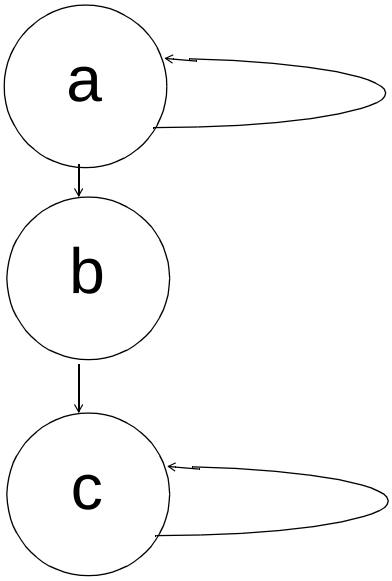


but for two processors it should be compiled as

```
cobegin
      do i=1,n
            a
            b
      end do
//
      do i=1,n
            c
            d
      end do
coend
```

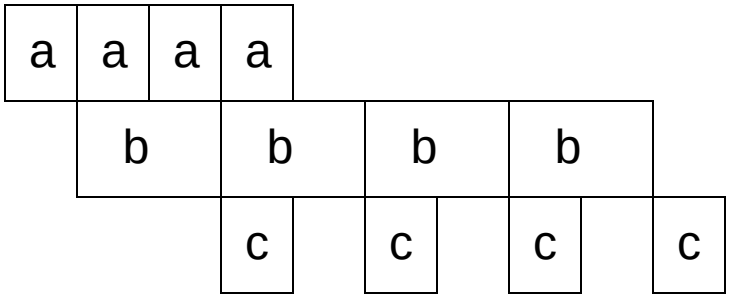


Example 2: The loop



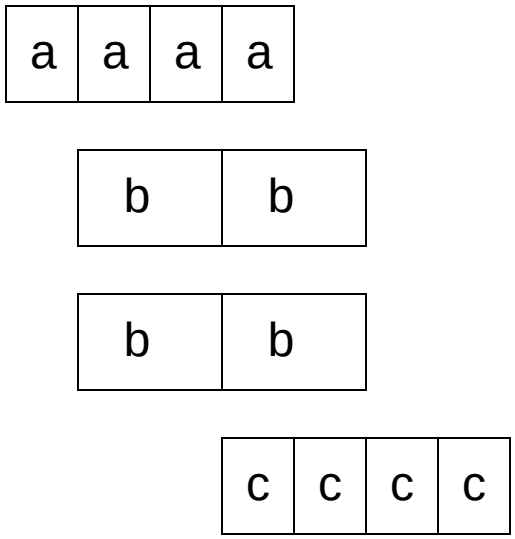
can be translated into

```
cobegin
      do i=1,n
            a
      end do
//
      do i=1,n
            b
      end do
//
      do i=1,n
            c
      end do
coend
```



or into

```
cobegin
      do i=1,n
            a
      coend
//
      do i=1,n,2
            cobegin
                  b
            //
                  b
            coend
      end do
//
      do i=1,n
            c
      end do
coend
```

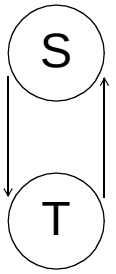


- If the execution time of b is unknown, (e.g. it includes a while loop), it is not possible to decide at compile-time how many copies of b to do in parallel.

2. Cycles force sequential execution

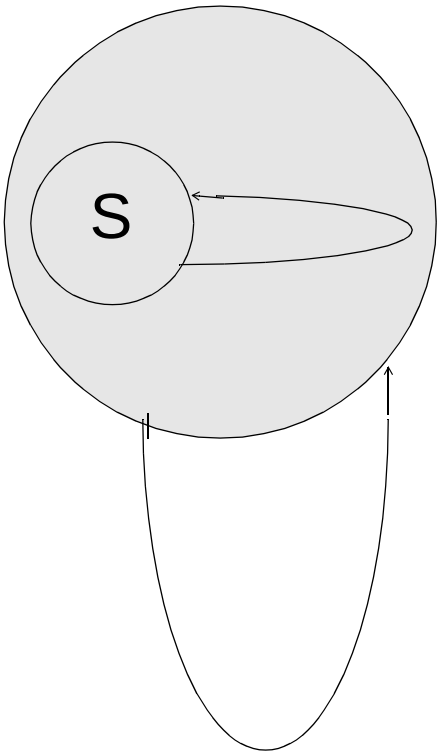
Example 3

```
do i=3,n
S:   a(i)=b(i-2)-1
T:   b(i)=a(i-3)*k
end do
```



Example 4

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



9.17 Cyclic dependences -- DOACROSS

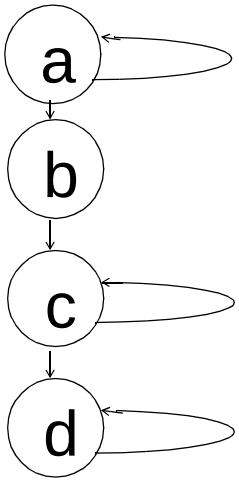
A loop with cyclic dependences can be transformed into DOACROSS as shown next:

```
      do i=1,n
            a(i) = b(i) + a(i-1)
            c(i) = a(i) + c(i-1)
      end do
                  ↓
c$doacross order(aa,bb),share(a,b,c)
      do i=1,n
c$order aa
            a(i) = b(i) + a(i-1)
c$endorder aa
c$order cc
            c(i) = a(i) + c(i-1)
c$endorder cc
      end do
```

DOACROSS has the advantage that all implicit tasks esecute the same code. This facilitates code assignment.

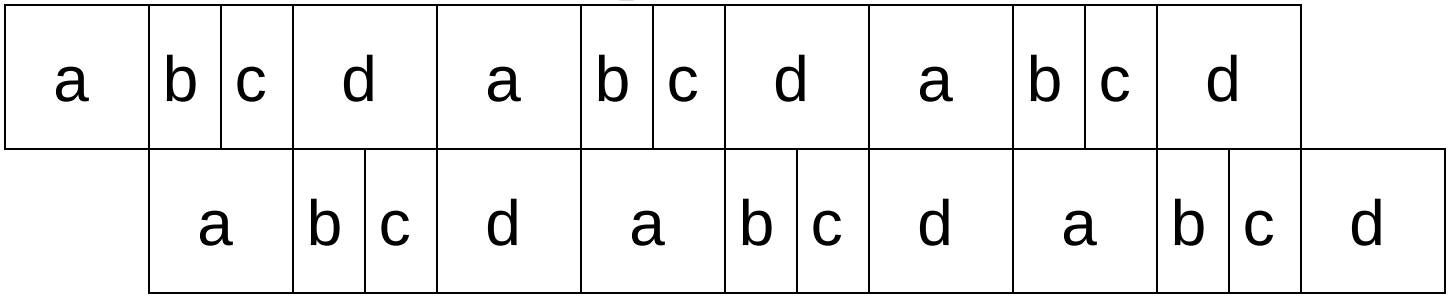
Other advantage of the DOACROSS construct over the DOPIPE construct are illustrated in the following examples.

Example 1:

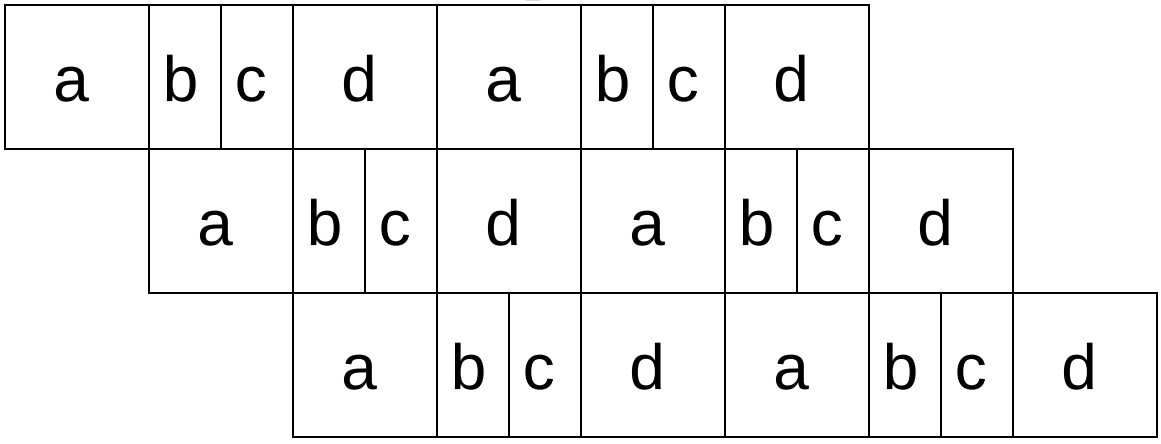


The same translation works for two or three processors:

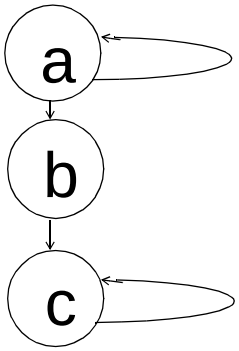
Two processors

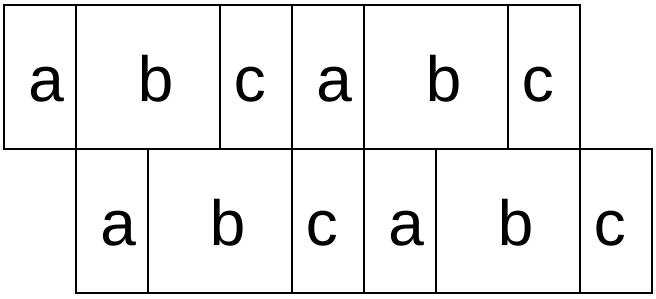


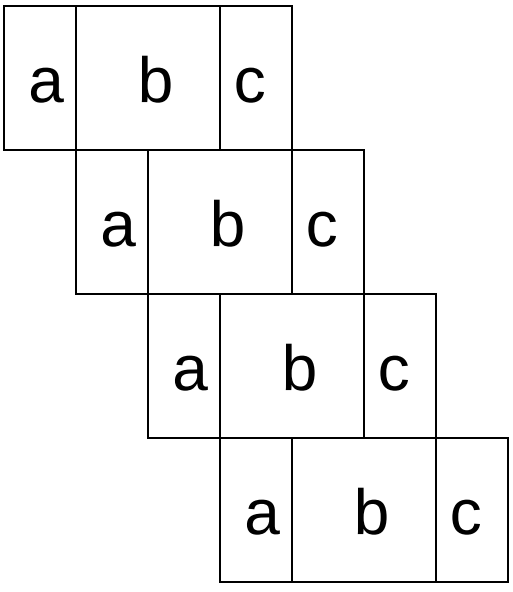
Three processors



Example 2:







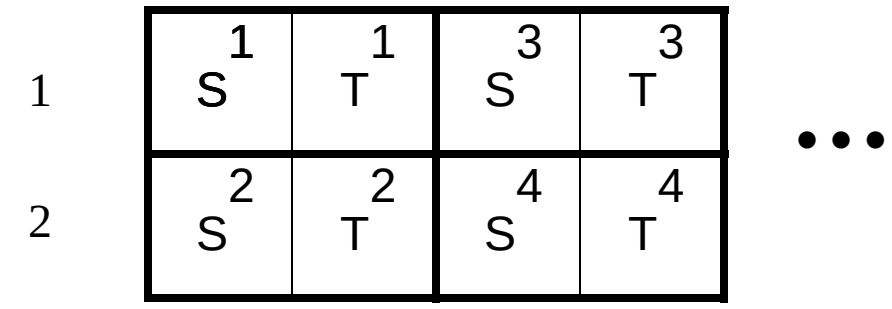
- Increasing the number of processors improve performance

Example 3

When the following loop is executed as a doacross on two processors

```
do i=1,n
S:    a(i) = b(i-2) -1
T:    b(i) = a(i-3) * k
end do
```

we get the following time lines (S^i stands for statement S in iteration i)



Cycle shrinking takes place automatically.

This is also true in the case of multiply-nested loops where all what is needed is to use a tuple as the loop index as in

doacross (i,j,k)=[1..n_1].c.[1..n_2].c.[1..n_3]

Example 4:

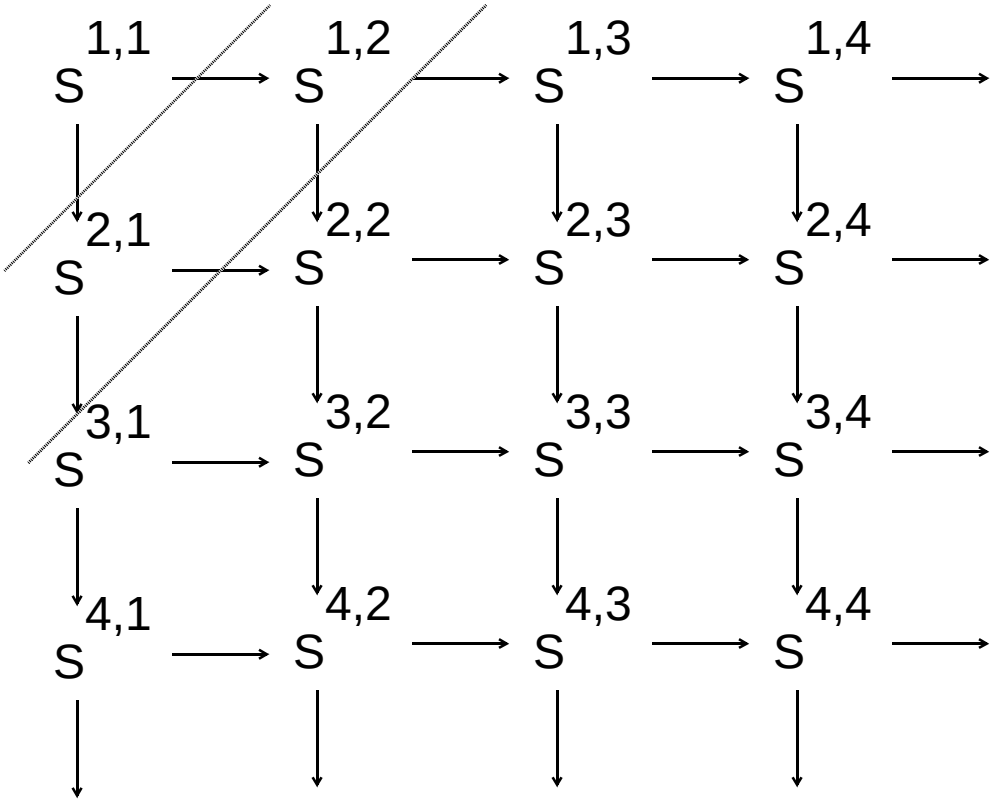
The following loop

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

can be translated into the following doacross loop:

```
doacross (i,j) = [1..n].c.[1..n]
      wait (ev(i-1,j)); wait (ev(i,j-1))
S:    a(i,j) = a(i-1,j) + a(i.j-1)
      post (ev(i,j))
end doacross
```

The iteration space of the previous loop is:



and its time lines when executed on n processors are: