Chapter 9:

DEPENDENCE-DRIVEN LOOP MANIPULATION
9.1 DEPENDENCES

Flow Dependence (True Dependence)

\[ S_1 \quad X = A + B \]
\[ S_2 \quad C = X + 1 \]

Anti Dependence

\[ S_1 \quad A = X + B \]
\[ S_2 \quad X = C + D \]

Output Dependence

\[ S_1 \quad X = A + B \]
\[ \ldots \]
\[ S_2 \quad X = C + D \]
9.2 DEPENDENCE AND PARALLELIZATION (SPREADING)

\[ S_1; S_2; S_3 \text{ can execute in parallel with } S_4; S_5; S_6 \]
\[ S_8; S_9 \text{ " " " " " " } S_{10}; S_{11} \]
C$OMP PSECTIONS
C$OMP SECTION
   S1
   S2
   S3
C$OMP SECTION
   S4
   S5
   S6
C$OMP END PSECTIONS
   S7
C$OMP PSECTIONS
C$OMP SECTION
   S8
   S9
C$OMP SECTION
   S10
   S11
C$OMP END PSECTIONS
9.3 RENAMING

(To remove memory-related dependences)

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 \( X(I+1) = B(I) + 1 \)
S2 \( A(I) = X(I) \)
END DO

DO I=1,N
S1 \( X(I) = B(I) + 1 \)
S2 \( A(I) = X(I+1) + 1 \)
END DO
9.6 DEPENDENCE ANALYSIS

\[
\begin{align*}
\text{DO I=1,N} \\
S1 & \quad X(F(I)) = B(I) + 1 \\
S2 & \quad A(I) = X(G(I)) + 2 \\
\text{END DO}
\end{align*}
\]

We say that

\[
\begin{align*}
\text{IFF } & \quad \exists I_1 \leq I_2 \\
\exists & \quad F(I_1) = G(I_2) \\
\text{[ALSO } & \quad I_1, I_2 \in [1, N]\text{]}
\end{align*}
\]

We say that

\[
\begin{align*}
\text{IFF } & \quad \exists I_1 < I_2 \\
\exists & \quad F(I_2) = G(I_1)
\end{align*}
\]
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 PARALLEL DO I=1,N
  A(1:N)=X(1:N)+1 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.

\[
\begin{align*}
  \text{DO } & I=1, N \\
  & A(I) = A(I-1) + B(I) \\
  \text{END DO}
\end{align*}
\]

\[
A(1:N) = \text{RECI}(B(1:N), A(0), N)
\]

\[
\begin{align*}
  X &= A(1) \\
  \text{DO } & I=2, N \\
  & \text{IF } (X > A(I)) X = A(I) \\
  \text{END DO}
\end{align*}
\]

\[
X = \text{MIN}(A(1:N))
\]
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
    X = A(I)
END IF
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 determines whether or not the loop headers can be interchanged.
- For example, the following loop headers can be interchanged

\[
\begin{align*}
do & \quad i=1, n \\
& \quad \begin{align*}
do & \quad j=1, n \\
& \quad a(i,j) = a(i,j-1) + a(i-1,j) \\
& \quad \end{align*} \\
end do \\
end do
\end{align*}
\]

- 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+1)
    end do
end do
9.11 DEPENDENCE REMOVAL

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

Scalar Expansion:

\[
\begin{align*}
\text{DO} & \quad I=1,N \\
S1: & \quad A=B(I)+1 \\
S2: & \quad C(I)=A+D(I) \\
\text{END DO}
\end{align*}
\]

\[
\begin{align*}
\text{DO} & \quad I=1,N \\
S1: & \quad A1(I)=B(I)+1 \\
S2: & \quad C(I)=A1(I)+D(I) \\
\text{END DO} \\
A & =A1(N)
\end{align*}
\]
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:

\[
\begin{align*}
\text{do } & i=1,n \\
S_1 & : a(i) = b(i) + c(i) \\
S_2 & : d(i) = x(i) + 1
\end{align*}
\]

end do

Example 2:

\[
\begin{align*}
\text{do } & i=1,n \\
S_1 & : a(i) = b(i) + c(i) \\
S_2 & : d(i) = a(i) + 1
\end{align*}
\]
Example 3:

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

\[
\begin{align*}
\text{do i=1,n} & & \quad S_1 \\
& & \downarrow 1 \\
& & S_2 \\
\text{do i=1,n} & & \quad S_1 \\
& & \downarrow 0 \\
& & S_2
\end{align*}
\]
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:

\[
\begin{align*}
\text{do } & i=1,n \\
S & : a(i) = b(i) + c(i-1) \\
1 & \\
S & : c(i) = d(i) \\
2 & \\
\text{end do}
\end{align*}
\]

Performing alignment without sorting first will clearly be incorrect in this case.
Another method for eliminating cross-iteration dependences is to perform loop distribution.

Example:

```plaintext
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 Loop Coalescing for DOALL loops

- A perfectly nested DOALL loop such as

\[
\begin{align*}
doall & \ i=1,n1 \\
doall & \ j=1,n2 \\
doall & \ k=1,n3 \\
& \quad \ldots \\
end \ doall \\
end \ doall \\
end \ doall
\end{align*}
\]

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

\[
\begin{align*}
doall & \ (i,j,k) = (1..n1).c.(1..n2).c.(1..n3) \\
& \quad \ldots \\
end \ doall
\end{align*}
\]

This coalescing transformation is convenient for scheduling and could reduce the overhead involved in starting DOALL loops.
If the loop construct has only one dimension, coalescing can be done by creating a mapping from a single index, say x into a multimentional index.
9.15 *Cyclic Dependences -- DOPIPE*

- Assume a loop with two or more dependence cycles (strongly connected components or \( \pi \)-blocks)

- The first approach developed for concurrentization of do loops is illustrated below:

```plaintext
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)
    post(s)
  end do

//
do i=1,n
  wait(s)
  c(i) = a(i) + c(i-1)
end do
coend
```

i.e. to take a loop with two or more \( \pi \)-blocks such as:
and execute collections of $\pi$-blocks on separate processors in a pipelined fashion:
9.15.1 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

\[ T(a) + T(b) + nT(c) + T(d) + T(e) \]
9.15.2 DOPIPE and Loop Distribution

Assume a loop with the dependence graph shown on the right.

The loop could be distributed to produce:

```fortran
  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:
However, executing the original loop as a DOPIPE produces the same execution time with fewer processor (if
\[ \max(T(a), T(b)) \leq \max(T(c), T(d)) \] and the number of iterations >4):
9.15.3 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

\[\begin{array}{ccc}
   a & a & a \\
   b & c & b & c & b & c \\
   d & d & d \\
\end{array}\]
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

\begin{align*}
\text{cobegin} \\
\text{do i}=1, n \\
\hspace{1cm} a \\
\text{coend} \\
\mathllap{\text{//}} \\
\text{do i}=1, n, 2 \\
\hspace{1cm} \text{cobegin} \\
\hspace{3cm} b \\
\mathllap{\text{//}} \\
\hspace{3cm} b \\
\text{coend} \\
\text{end do} \\
\mathllap{\text{//}} \\
\text{do i}=1, n \\
\hspace{1cm} c \\
\text{end do} \\
\text{coend}
\end{align*}

• If the execution time of \( b \) is unknown, (e.g. it includes a \texttt{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
9.16 Cyclic dependences -- DOACROSS

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

```plaintext
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 execute the same code. This facilitates code assignment. Other advantage of the DOACROSS construct over the DOPIPE construct are illustrated in the following examples.
An alternative form of the doacross loop is:

\begin{verbatim}
  do i=1,n
    a(i) = b(i) + a(i-1)
    c(i) = a(i) + c(i-1)
  end do

  ↓

  post [s1(0)]
  post [s2(0)]

  c$ parallel do
    do i=1,n
      wait [s1(i-1)]
      a(i) = b(i) + a(i-1)
      post [s1(i)]
    wait [s2(i-1)]
    c(i) = a(i) + c(i-1)
    post [s2(i)]
  end do
\end{verbatim}
Example 1:

The same translation works for two or three processors

Two processors

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Three processors

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

© David A. Padua, 1998
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 stands for statement S in iteration i)

<table>
<thead>
<tr>
<th>Proc</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

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]c.[1..n]c.[1..n]
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:

\[
\text{c$\parallel$ parallel do} \\
do i=1,n \\
    do j=1,n \\
        wait (ev(i-1,j)) \\
        S: \quad a(i,j) = a(i-1,j) + a(i,j-1) \\
        post (ev(i,j))
    end do \\
end do
\]
The iteration space of the previous loop is:

and its time lines when executed on $n$ processors are:
9.17 Stripmining

A common transformation is the following:

```plaintext
do i=1,n
   ...
end do
↓
do l=1,n,P
do i=i,min(n,i+P-1)
   ...
end do
end do
```

This transformation is always correct.

It has several uses. One of them is to reduce synchronization costs (at the expense of parallelism) in dopipe and doacross loops.

Reduction of synchronization costs with dopipe is clear from the following example:
do i=1,n
    a(i) = b(i) + a(i-1)
    c(i) = a(i) + c(i-1)
end do
↓
cobegin
do l=1,n,P
    do i=l,min(l+P-1,n)
        a(i) = b(i) + a(i-1)
    end do
    Post(σ)
end do
//
do i=1,n
    Wait(σ)
    do i=l,min(l+P-1,n)
        c(i) = a(i) + c(i-1)
    end do
end do
coend
9.18 Run-time decisions

Sometimes all what is needed for translation into DOALL is a critical section.

The following loop has a cyclic dependence graph (output dependences)

\[
\text{do } i=1,n \\
\quad a(k(i)) = a(k(i)) + 1 \\
\text{end do}
\]

This loop can be transformed into DOALL by just inserting a critical section as shown next:

\[
\text{do } i=1,n \\
\quad\text{critical } a(k(i)) \text{ do} \\
\quad\quad a(k(i)) = a(k(i)) + 1 \\
\quad\text{end critical} \\
\text{end do}
\]
9.18.1 Handling Output Dependences at Run-Time

do i=1,n
    a(k(i)) = c(i) + 1
end do

To parallelize the following loop we create a structure for each \( a(i) \) with two components \%sync and \%data and translate into:

\[
\begin{align*}
    a(k(:))%sync &= 0 \\
    \text{doall } i=1,n \\
    \quad \text{critical } a(k(i)) \\
    \quad \quad \text{if } a(b(i))%sync < i \text{ then} \\
    \quad \quad \quad a(k(i))%data &= c(i) + 1 \\
    \quad \quad \quad a(k(i))%sync &= i \\
    \quad \quad \end{align*}
\]

end if
end critical
end doall
Assume \( k(i) \) has the following values

\[
i = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

\[
k(i) = \begin{array}{cccccccccc}
3 & 5 & 7 & 3 & 4 & 5 & 6 & 9 & 10 & 3 \\
\end{array}
\]

The critical section will be reached by all iterations. Let us assume the following order of arrival among the conflicting ones:

for \( a(3) \): 4 1 10

for \( a(5) \): 6 2

In the previous loop \( a(k(i)) \% data \) will be assigned only once for \( i=3, 5, 7, 8, 9, 10 \)

\( a(5) \) will be assigned once since when iteration 2 enters the critical section after iteration 6 leaves, \( a(5) \% sync \) will be 6, and the boolean function inside the if will be false.

\( a(3) \) will be assigned twice. Once for iteration 4 and once for iteration 10. No assignment takes place when iteration 1 enters the critical section after iteration 4 leaves.
9.18.2 Handling flow dependences at Run-Time

do i=1,n
   a(k(i)) = ...
   ... = a(j(i))
end do
↓
repeat until all(done)
    doall i=1,n
        if (.not.done(i)) then
            a(k(i))%sync = ∞
            a(j(i))%sync = ∞
        end if
    end doall
    doall i=1,n
        if (.not.done(i)) then
            critical a(k(i))
                if a(k(i))%sync > i then a(k(i))%sync=i
                if a(j(i))%sync > i then a(j(i))%sync=i
            end critical
        end if
    end doall
    doall i=1,n
        if (.not.done(i)) then
            if (a(k(i))%sync = i & a(j(i))%sync = i) then
                a(k(i)) = ...
                ...
            = a(j(i))
            done(i) = .true.
        end if
    end if
end doall
end repeat
On each iteration of the repeat, the second doall selects those iterations not processed, and for a collection of iterations $i_1, i_2, ... i_K$ with $k(i_1) = k(i_2) = ... = k(i_K) = K$

$$a(K) \% \text{sync} \text{ gets the value } \min(i_1, i_2, ... i_K).$$

The third doall computes only those pairs where both $a(k(i)) \% \text{sync}$ and $a(j(i)) \% \text{sync}$ have the same value. The reason these iterations can be executed is that either no previous iterations of the original do loop reference the same array elements or earlier iterations referring to the same elements of $a$ have already been executed in previous iterations of the repeat.