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)

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

$S_8; S_9$ can execute in parallel with $S_{10}; S_{11}$
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 \[ S_1 \rightarrow S_2 \] \hspace{1cm} \text{IFF } \exists I_1 \leq I_2 \\
\hspace{3cm} \exists F(I_1) = G(I_2) \\
\hspace{3cm} [\text{ALSO } I_1, I_2 \in [1,N]]

We say that \[ S_1 \rightarrow S_2 \] \hspace{1cm} \text{IFF } \exists I_1 < I_2 \\
\hspace{3cm} \exists 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.

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

\[
\begin{align*}
& \text{X(1:N)=B(1:N)+1} \\
& \text{A(1:N)=X(1:N)+1}
\end{align*}
\]

- 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)=RECIPE(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 \textbf{LOOP DISTRIBUTION}

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

\[
\begin{align*}
\text{DO } & I=1, N \\
S1: & \quad A(I) - B(I) + C(I) \\
S2: & \quad D(I) = D(I-1) + A(I) \\
S3: & \quad \text{IF } (X.GT.A(I)) \text{ THEN} \\
S4 & \quad \text{X} = A(I) \text{ ENDIF}
\end{align*}
\]

\[
\begin{align*}
\text{END DO}
\end{align*}
\]

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

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

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

\[
\begin{align*}
\text{END DO}
\end{align*}
\]
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

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

- However, the headers in the following loop cannot be interchanged
\begin{verbatim}
do i=1,n
    do j=1,n
        a(i,j) = a(i,j-1) + a(i-1,j)
    end do
end do
\end{verbatim}
9.11 DEPENDENCE REMOVAL

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

Scalar Expansion:

\[
\begin{align*}
\text{DO } & \quad \text{I}=1, N \\
\text{S1: } & \quad A = B(I) + 1 \\
\text{S2: } & \quad C(I) = A + D(I) \\
\text{END DO } & \\
\text{DO } & \quad \text{I}=1, N \\
\text{S1: } & \quad A_1(I) = B(I) + 1 \\
\text{S2: } & \quad C(I) = A_1(I) + D(I) \\
\text{END DO } & \\
A = A_1(N)
\end{align*}
\]
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:

```
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)
  S1 : d(i) = x(i) + 1
end do
```
Example 3:

\begin{verbatim}
  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
\end{verbatim}
• When there are cross iteration dependences, but no cycles, do loops can be *aligned* to be transformed into DOALLs

Example 1:

```plaintext
do i=1,n
  \( S_1 \) : a(i) = b(i) + 1
  \( S_2 \) : c(i) = a(i-1)^2
end do
```

↓

```plaintext
do i=0,n
  \( S_1 \) : if i>0 then a(i) = b(i) + 1
  \( S_2 \) : if i<n then c(i+1) = a(i)^2
end do
```
• Sometimes we have to replicate to achieve alignment

Example 2:

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

...end do

\[\downarrow\]

\[
\begin{align*}
\text{do } i &= 1, n \\
    a(i) &= b(i) + c(i) \\
    a1(i) &= b(i) + c(i) \\
    d(i) &= a1(i) + a(i-1) \\
\end{align*}
\]

...end do

\[\downarrow\]

\[
\begin{align*}
\text{do } i &= 0, n \\
    \text{if } i > 0 \text{ then } a(i) &= b(i) + c(i) \\
    \text{if } i < n \text{ then } a1(i+1) &= b(i+1) + c(i+1) \\
    \text{d}(i+1) &= a1(i+1) + a(i) \\
\end{align*}
\]

...end do
• Need for replication could propagate.

Example 3:

```plaintext
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:

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

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:

\[
\begin{align*}
S_1 &: a(1) = b(1) + c(0) \\
\text{do } i &= 1, n-1 \\
S_2 &: c(i) = d(i) \\
S_1 &: a(i+1) = b(i+1) + c(i) \\
S_2 &: c(n) = d(n)
\end{align*}
\]
• Another method for eliminating cross-iteration dependences is to perform loop distribution.

Example:

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

\[
\begin{align*}
downarrow
\end{align*}
\]

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

1. Loops with a cycle-free dependence graphs can be easily vectorized.

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

   $$\text{do } i=1,n$$
   $$\text{S} : a(i) = b(i) + c(i)$$
   $$\quad \text{1}$$
   $$\text{S} : d(i) = a(i-1) + x(i)$$
   $$\quad \text{2}$$
   $$\text{S} : e(i) = d(i) + t$$
   $$\quad \text{3}$$
   $$\text{end do}$$

   ↓

   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
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.

\[
\begin{align*}
  \text{do } & i=1, n \\
  S_1 & : a(i) = b(i-1) + d(i-1) \\
  S_2 & : b(i) = t + x(i) \\
  S_3 & : d(i) = d(i-1) + y(i) \\
  \text{end do}
\end{align*}
\]

\[
\begin{align*}
  d(1:n) &= \text{lr1}(d(0), y(1:n)) \\
  b(1:n) &= x(1:n) + t \\
  a(1:n) &= b(0:n-1) + d(0:n-1)
\end{align*}
\]
9.15 Loop Coalescing for DOALL loops

- A perfectly nested DOALL loop such as

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

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

```plaintext
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 $\pi$-blocks)
- The first approach developed for concurrentization of do loops is illustrated below:

```markdown
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(\sigma)
    end do
    P(\sigma)
    c(i) = a(i) + c(i-1)
end do
coend
```
i.e. to take a loop with two or more $\pi$-blocks such as:

![Diagram of loop with two or more $\pi$-blocks](image)

and execute collections of $\pi$-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:
However, executing the original loop as a DOPIPE produces the same execution time with fewer processor (if number of iterations >4):
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

coen
Example 2: The loop can be translated into

\[
\text{cobegin}
\begin{align*}
\text{do } i &= 1, n \\
&\quad \text{a} \\
&\quad \text{end do} \\
// \\
\text{do } i &= 1, n \\
&\quad \text{b} \\
&\quad \text{end do} \\
// \\
\text{do } i &= 1, n \\
&\quad \text{c} \\
&\quad \text{end do}
\end{align*}
\text{coend}
\]
or into

cobegin
do i=1,n
  a
coend

//
doi=1,n,2
  cobegin
    b
  //
    b
coend
end do

//
doi=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

\[
\text{do } i=3, n \\
\text{S: } a(i) = b(i-2) - 1 \\
\text{T: } b(i) = a(i-3) \times k \\
\text{end do}
\]

Example 4

\[
\text{do } i=1, n \\
\quad \text{do } j=1, n \\
\text{S: } a(i, j) = a(i-1, j) + a(i, j-1) \\
\text{end do} \\
\text{end do}
\]
9.17  *Cyclic dependences -- DOACROSS*

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

\[
\begin{align*}
d\ i=1,n \\
&\quad a(i) = b(i) + a(i-1) \\
&\quad c(i) = a(i) + c(i-1) \\
end\ do \\
\downarrow \\
c\$doacross\ order(aa,bb),share(a,b,c) \\
&\quad do\ i=1,n \\
&\quad c\$order\ aa \\
&\quad \quad a(i) = b(i) + a(i-1) \\
&\quad c\$endorder\ aa \\
&\quad c\$order\ cc \\
&\quad \quad c(i) = a(i) + c(i-1) \\
&\quad c\$endorder\ cc \\
end\ do
\end{align*}
\]

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.
Example 1:

The same translation works for two or three processors:

**Two processors**

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</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>
</tbody>
</table>
```

**Three processors**

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</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>
```
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:  \quad a(i) = b(i-2) - 1
T:  \quad b(i) = a(i-3) * k
end do

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

Proc.

\[
\begin{array}{cccc}
1 & 1 & 3 & 3 \\
S & T & S & T \\
2 & 2 & 4 & 4 \\
S & T & S & T \\
\end{array}
\]

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

\[
\text{doacross } (i,j,k)=\left[1..n^1\right].c.\left[1..n^2\right].c.\left[1..n^3\right]
\]
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: