Loop-independent vs. loop-carried dependences

[§3.2] Loop-carried dependence: dependence exists across iterations; i.e., if the loop is removed, the dependence *no longer exists.*

Loop-independent dependence: dependence exists within an iteration; i.e., if the loop is removed, the dependence still exists.

*Example:*

```
for (i=1; i<n; i++) {
    S1: a[i] = a[i-1] + 1;
    S2: b[i] = a[i];
}

for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        S4: a[i][j] = a[i-1][j] + 1;
```

*Iteration-space Traversal Graph (ITG)*

[§3.2.1] The ITG shows graphically the order of traversal in the iteration space. This is sometimes called the *happens-before relationship.* In an ITG,

- A *node* represents a point in the iteration space
- A *directed edge* indicates the next point that will be encountered after the current point is traversed

*Example:*

```
for (i=1; i<4; i++)
    for (j=1; j<4; j++)
        S3: a[i][j] = a[i][j-1] + 1;
```
Loop-carried Dependence Graph (LDG)

- LDG shows the true/anti/output dependence relationship graphically.
- A node is a point in the iteration space.
- A directed edge represents the dependence.

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```
Another example:

```c
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
    {
        S2: a[i][j] = b[i][j] + c[i][j];
        S3: b[i][j] = a[i][j-1] * d[i][j];
    }
```

- Draw the ITG
- List all the dependence relationships

Note that there are two “loop nests” in the code.

- The first involves S1.
- The other involves S2 and S3.

What do we know about the ITG for these nested loops?
Dependence relationships for Loop Nest 1

• True dependences:
  o $S_1[i,j] \rightarrow T S_1[i,j+1]
  o $S_1[i,j] \rightarrow T S_1[i+1,j]

• Output dependences:
  o None

• Anti-dependences:
  o $S_1[i,j] \rightarrow A S_1[i+1,j]
  o $S_1[i,j] \rightarrow A S_1[i,j+1]

Exercise: Suppose we dropped off the first half of $S_1$, so we had

$S_1$: $a[i][j] = a[i-1][j] + a[i+1][j]$;

or the last half, so we had

$S_1$: $a[i][j] = a[i][j-1] + a[i][j+1]$;

Which of the dependences would still exist?
Draw the LDG for Loop Nest 1.

\[
\begin{array}{cccc}
1 & 2 & \ldots & n \\
1 & & & \\
i & 2 & & \\
\ldots & & & \\
n & & & \\
\end{array}
\]

Note: each edge represents both true, and anti-dependences.

Dependence relationships for Loop Nest 2

- True dependences:
  - \(s_2[i, j] \rightarrow_T s_3[i, j+1]\)

- Output dependences:
  - None

- Anti-dependences:
  - \(s_2[i, j] \rightarrow_A s_3[i, j]\) (loop-independent dependence)
Draw the LDG for Loop Nest 2.

Why are there no vertical edges in this graph? Answer here.

Why is the anti-dependence not shown on the graph?

Finding parallel tasks across iterations

[§3.2.2] Analyze loop-carried dependences:

- Dependences must be enforced (especially true dependences; other dependences can be removed by privatization)
- There are opportunities for parallelism when some dependences are not present.

Example 1

```
for (i=2; i<=n; i++)
  S: a[i] = a[i-2];
```

LDG:
We can divide the loop into two parallel tasks (one with odd iterations and another with even iterations): 

**Example 2**

\[
\begin{align*}
&\text{for } (i=0; i<n; i++) \\
&\quad \text{for } (j=0; j< n; j++) \\
&\quad \text{S3: } a[i][j] = a[i][j-1] + 1;
\end{align*}
\]

LDG

\[
\begin{array}{cccc}
1 & 2 & \ldots & n \\
\downarrow & \downarrow & \ldots & \downarrow \\
1 & 2 & \ldots & n
\end{array}
\]

How many parallel tasks are there here?

**Example 3**

\[
\begin{align*}
&\text{for } (i=1; i<=n; i++) \\
&\quad \text{for } (j=1; j<=n; j++) \\
&\quad \text{S1: } a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];
\end{align*}
\]

LDG

\[
\begin{array}{cccc}
1 & 2 & \ldots & n \\
\downarrow & \downarrow & \ldots & \downarrow \\
1 & 2 & \ldots & n
\end{array}
\]

Note: each edge represents both true, and anti-dependences
Identify which nodes are not dependent on each other

In each anti-diagonal, the nodes are independent of each other

We need to rewrite the code to iterate over anti-diagonals:

Calculate number of anti-diagonals

for each anti-diagonal do

Calculate the number of points in the current anti-diagonal

for each point in the current anti-diagonal do

Compute the value of the current point in the matrix

Parallelize loops highlighted above.

```c
for (i=1; i <= 2*n-1; i++) {// 2n-1 anti-diagonals
    if (i <= n) {
        points = i; // number of points in anti-diag
        row = i; // first pt (row,col) in anti-diag
        col = 1; // note that row+col = i+1 always
    } else {
        points = 2*n – i;
        row = n;
        col = i-n+1; // note that row+col = i+1 always
    }
    for_all (k=1; k <= points; k++) {
        a[row][col] = … // update a[row][col]
        row--; col++;
    }
}
```
DOACROSS Parallelism

[§3.2.3] Suppose we have this code:

Can we execute anything in parallel?

Well, we can’t run the iterations of the for loop in parallel, because …

\[ S[i] \rightarrow T S[i+1] \] (There is a loop-carried dependence.)

But, notice that the \( b[i] \ast c[i] \) part has no loop-carried dependence.

This suggests breaking up the loop into two:

\[
\begin{align*}
\text{for } (i=1; i<=N; i++) \{ \\
\quad S1: \text{temp}[i] = b[i] \ast c[i]; \\
\quad S2: a[i] = a[i-1] + \text{temp}[i]; \\
\}
\end{align*}
\]

The first loop is \( || \)zable. The second is not.

Execution time: \( N \times (T_{S1} + T_{S2}) \)

What is a disadvantage of this approach?

Here’s how to solve this problem:

\[
\begin{align*}
\text{post}(0); \\
\text{for } (i=1; i<=N; i++) \{ \\
\quad S1: \text{temp} = b[i] \ast c[i]; \\
\quad \text{wait}(i-1); \\
\quad S2: a[i] = a[i-1] + \text{temp}; \\
\quad \text{post}(i); \\
\}
\end{align*}
\]

What is the execution time now?

Parallelism across statements in a loop

- [§3.2.4] Identify dependences in a loop body.
- If there are independent statements, can split/distribute the loops.
Example:

```c
for (i=0; i<n; i++) {
    S1: a[i] = b[i+1] * a[i-1];
    S2: b[i]  = b[i] * coef;
    S3: c[i] = 0.5 * (c[i] + a[i]);
    S4: d[i] = d[i-1] * d[i];
}
```

Loop-carried dependences:

```
for (i=0; i<n; i++) {
    S1: a[i] = b[i+1] * a[i-1];
    S2: b[i]  = b[i] * coef;
    S3: c[i] = 0.5 * (c[i] + a[i]);
    S4: d[i] = d[i-1] * d[i];
}
```

Loop-indep. dependences:

```
for (i=0; i<n; i++) {
    S4: d[i] = d[i-1] * d[i];
}
```

Note that S4 has no dependences with other statements

“$S1[i] \rightarrow A S2[i+1]$” implies that S2 at iteration $i+1$ must be executed after S1 at iteration $i$. Hence, the dependence is not violated if all S2s executed after all S1s.

After loop distribution:

```
for (i=0; i<n; i++) {
    S1: a[i] = b[i+1] * a[i-1];
    S2: b[i]  = b[i] * coef;
    S3: c[i] = 0.5 * (c[i] + a[i]);
}
for (i=0; i<n; i++) {
    S4: d[i] = d[i-1] * d[i];
}
```

Each loop is a parallel task.

This is called function parallelism.

Further transformations can be performed (see p. 44 of text).

This is called function parallelism, and can be distinguished from data parallelism, which we saw in DOALL and DOACROSS.

Characteristics of function parallelism:

- 
- 

Can use function parallelism along with data parallelism when data parallelism is limited.

**DOPIPE Parallelism**

[§3.2.5] Another strategy for loop-carried dependences is pipelining the statements in the loop.
Consider this situation:

Loop-carried dependences:

\[
\begin{align*}
&\text{for } (i=2; \ i<=N; \ i++) \ { } \\
&S1: \ a[i] = a[i-1] + b[i]; \\
&S2: \ c[i] = c[i] + a[i]; \\
&\text{post}(i);
\end{align*}
\]

Loop-indep. dependences:

\[
\begin{align*}
&\text{for } (i=2; \ i<=N; \ i++) \ { } \\
&a[i] = a[i-1] + b[i]; \\
&\text{post}(i);
\end{align*}
\]

To parallelize, we just need to make sure the two statements are executed in sync:

\[
\begin{align*}
&\text{for } (i=2; \ i<=N; \ i++) \ { } \\
&a[i] = a[i-1] + b[i]; \\
&\text{post}(i);
\end{align*}
\]

\[
\begin{align*}
&\text{for } (i=2; \ i<=N; \ i++) \ { } \\
&\text{wait}(i);
\end{align*}
\]

\[
\begin{align*}
&\text{for } (i=2; \ i<=N; \ i++) \ { } \\
&c[i] = c[i] + a[i];
\end{align*}
\]

Question: What’s the difference between DOACROSS and DOPIPE?