Finding parallel tasks across iterations

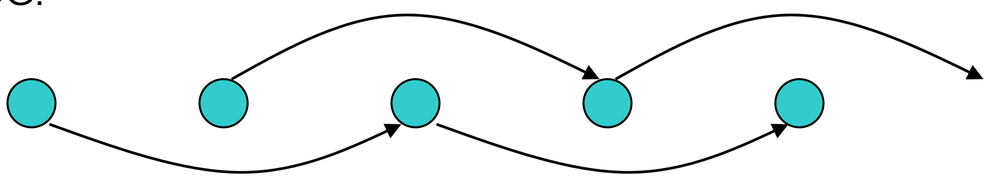
[§3.3.1] Analyze loop-carried dependences:

- 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:



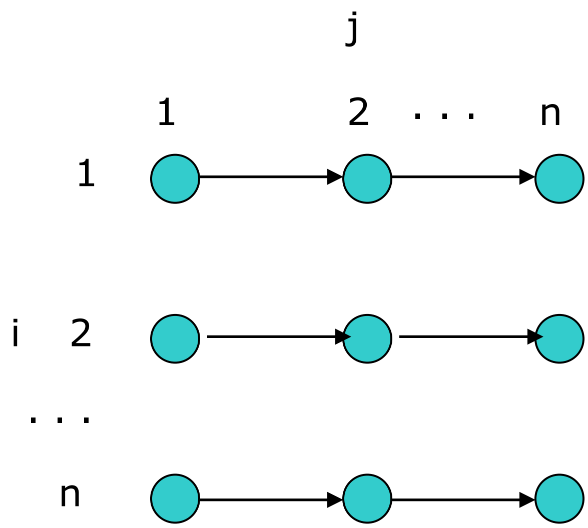
We can divide the loop into two parallel tasks (one with odd iterations and another with even iterations):

```
for (i=2; i<=n; i+=2)
  S: a[i] = a[i-2];
for (i=3; i<=n; i+=2)
  S: a[i] = a[i-2];
```

Example 2

```
for (i=0; i<n; i++)
  for (j=0; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

LDG

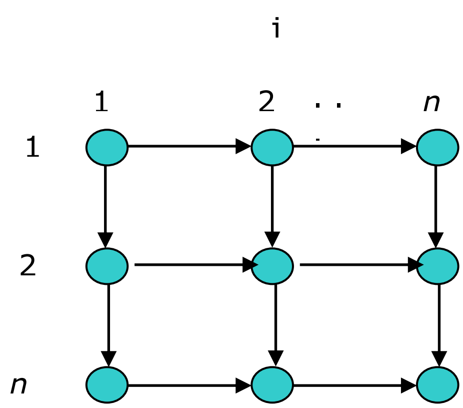


How many parallel tasks are there here?

Example 3

```
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];
```

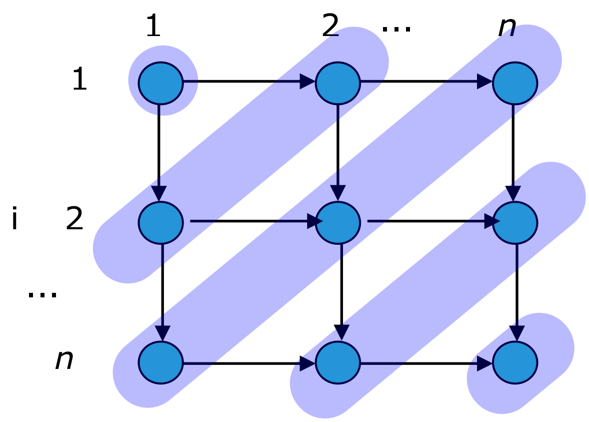
LDG



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



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

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_all points in the current anti-diagonal do
    Compute the value of the current point in the matrix

Parallelize the loops highlighted above.

```
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

[§3.3.2] Suppose we have this code:

Can we execute anything in parallel?

```
for (i=1; i<=N; i++) {
  S: a[i] = a[i-1] + b[i] * c[i];
}
```

Well, we can't run the iterations of the **`for`** loop in parallel, because …

`S[i]` →T **`S[i+1]`** (There is a loop-carried dependence.)

But, notice that the **`b[i]*c[i]`** part has no loop-carried dependence.

This suggests breaking up the loop into two:

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

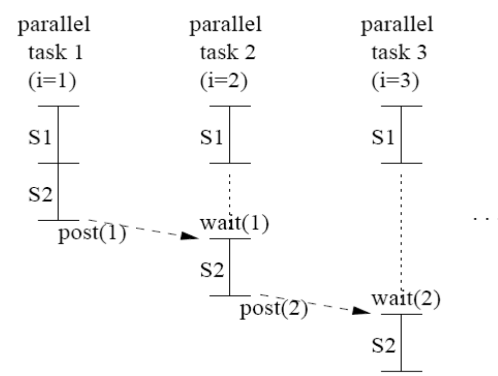
The first loop is ||izable.
The second is not.

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

What is a disadvantage of this approach?

Here's how to solve this problem:

```
post(0);
for_all (i=1; i<=N; i++) {
  S1: temp = b[i] * c[i];
  wait(i-1);
  S2: a[i] = a[i-1] + temp;
  post(i);
}
```



What is the execution time now?

Function parallelism

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

Example:

```
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:

Loop-indep. dependences:

Note that S4 has no dependences with other statements

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.*

It can be distinguished from *data parallelism*, which we saw in DOALL and DOACROSS.

Further transformations can be performed (see p. 64 of text).

"**`S1[i]`** →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 execute after all S1s.

Characteristics of function parallelism:

-
-

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

DOPIPE Parallelism

[§3.3.4] Another strategy for loop-carried dependences is pipelining the statements in the loop.

Consider this situation:

```
for (i=2; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  S2: c[i] = c[i] + a[i];
}
```

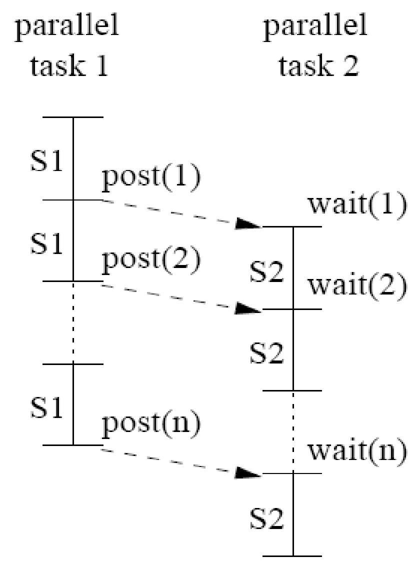
Loop-carried dependences:

Loop-indep. dependences:

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

```
for (i=2; i<=N; i++) {
  a[i] = a[i-1] + b[i];
  post(i);
}

for (i=2; i<=N; i++) {
  wait(i);
  c[i] = c[i] + a[i];
}
```



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

Determining variable scope

[§3.6] This step is specific to the shared-memory programming model. For each variable, we need to decide how it is used. There are three possibilities:

- Read-only: *variable is only read by multiple tasks*
- R/W non-conflicting: *variable is read, written, or both by only one task*
- R/W conflicting: *variable is written by one task and may be read by another*

Intuitively, why are these cases different?

Example 1

Let's assume each iteration of the **for** *i* loop is a parallel task.

```
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];
  }
```

Fill in the tableaus here.

Read-only	R/W non-conflicting	R/W conflicting

Now, let's assume that each **for** *j* iteration is a separate task.

Read-only	R/W non-conflicting	R/W conflicting

Do these two decompositions create the same number of tasks?

Example 2

Let's assume that each **for** *j* iteration is a separate task.

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

Read-only	R/W non-conflicting	R/W conflicting

Exercise: Suppose each **for** *i* iteration were a separate task …

Read-only	R/W non-conflicting	R/W conflicting

To test your knowledge of this approach, try the recent homework problem on the following page:

Problem *k*. *(15 points)* The following code is a commonly used algorithm in image processing applications.

Consider an image *f* with width=ImageWidth and height=ImageHeight. *f* is a 2D grid of pixels. *k* is a kernel of width=2*w*+1 and height=2*h*+1 where (2*w*+1) < ImageWidth and (2*h*+1) < ImageHeight. The image *f* is processed using the kernel *k* to produce a new image *g* as shown:

```
for y = 0 to ImageHeight do
    for x = 0 to ImageWidth do
        sum = 0
        for i= -h to h do
            for j = –w to w do
                sum = sum + k[j,i] * f[x – j, y – i]
            end for
        end for
        g[x y] = sum
    end for
end for
```

(a). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if the **for** *y* loop is parallelized.

Read only	R/W non-conflicting	R/W conflicting

(b). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if (only) the **for** *i* loop is parallelized. Assume that the **for** *i* tasks for the previous value of *x* must complete before the **for** *i* tasks of the current value of *x* are started.

Read only	R/W non-conflicting	R/W conflicting

(c). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if the **for** *i* loop is parallelized. Assume that the **for** *i* tasks for the previous value of *x* do not have to complete before the **for** *i* tasks of the current value of *x* are started.

Read only	R/W non-conflicting	R/W conflicting