Multicore Computing
Lecture06 – Loop Dependence

SUNG KYUN KWAN UNIVERSITY

남범석

bnam@skku.edu

Rule of Thumb

- A loop that matches the following can be parallelized
  - Without restructuring

- 1. All assignments are to arrays
- 2. Each element is assigned by at most one iteration
- 3. No iteration reads elements assigned by any other iteration

```
// simplest case
for (i=0; i<N; i++){
    C[i] = A[i] + B[i-1];
}
```

Loop-carried Dependences

- Q: Can we parallelize the following loop using OpenMP?

```
a[0] = 1;
for (i=1; i<N; i++){
    a[i] = a[i] + a[i-1];
}
```

```
i=1: a[1] = a[1] + a[0];
i=2: a[2] = a[2] + a[1];
```

```
// how about this?
for (i=0; i<N; i+=2){
    A[i] = A[i] + A[i-1];
}
```

```
// and this?
for (i=0; i<N/2; i+=2){
  A[idx[i]] = A[idx[i]]+B[idx[i]];
}
```

Detecting Dependence

- Analyze how each variable is used within a loop
- Is the variable only read and never written?
  - → No dependence
- For each variable written: can there be any accesses in other iterations than the current?
  - → There exist dependences

Data Dependence

- A *dependence* arises when
  - one operation depends on an earlier operation

- Executing Two Independent Statements
  - On one processor:
    - Statement 1;
Statement 2;
  - On two processors in parallel:
    - Processor 1: Processor 2:
Statement 1; Statement 2;
    - **Sequential consistency** is guaranteed if
      - Computation results are the same (independent of order)

Data Dependences

- Example 1

  S1: a=1;

  S2: b=1;

- ❒ Statements are independent

- Example 2

  S1: a=1;

  S2: b=2*a;

S_1 **x = …**

S_2 **… = x**

- ❒ Dependent (*true dependence*)
  - ❍ Second is dependent on first
  - ❍ Cannot remove dependency

- Example 3

  S1: a=b;

  S2: b=1;

S_1 **… = x**

S_2 **x = …**

- ❒ Dependent (*anti dependence*)
  - ❍ First is dependent on second
  - ❍ Can you remove dependency?

- Example 4

  S1: a=f(x);

  S2: a=b;

S_1 **x = …**

S_2 **x = …**

- ❒ Dependent (*output dependence*)
  - ❍ Second is dependent on first
  - ❍ Can you remove dependency?

Data Dependences

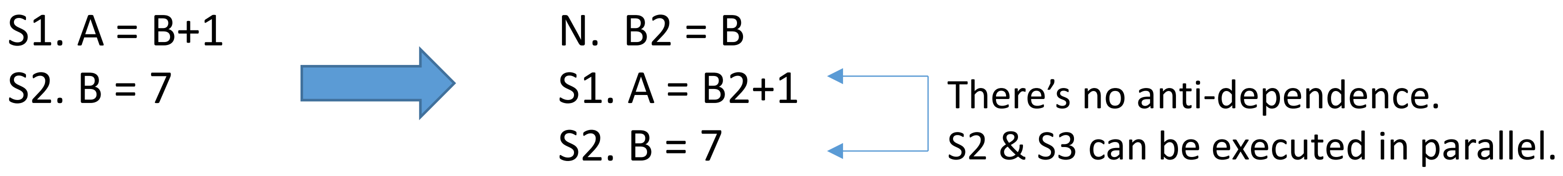
- True dependence
  - Read-After-Write (RAW)

- False dependence
  - Anti-dependence
    - Write-After-Read (WAR)
      - S1. A = B+1   // S1 needs an old value before S2 overwrites it.
      - S2. B = 7
  - Output dependence
    - Write-After-Write (WAW)
      - Ordering affects the final output value of a variable.

- Some dependences can be removed by
  - Rearranging statements
  - Eliminating statements
  - Rewriting with variable renaming

Data Dependences

- Anti-dependence
  - Write-After-Read (WAR)
    - Rewrite with variable renaming

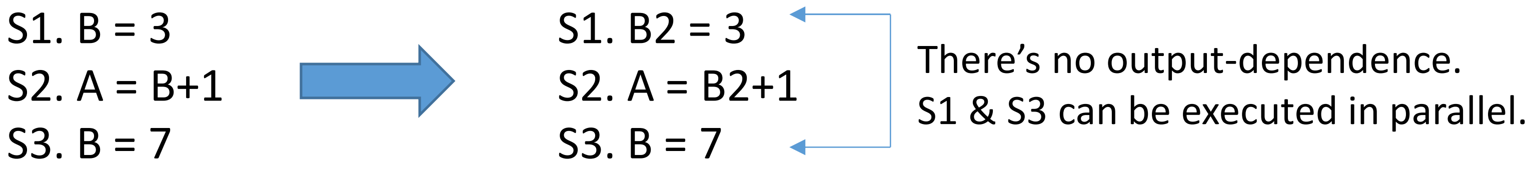


```
S1. A = B+1
S2. B = 7
```

```
N.  B2 = B
S1. A = B2+1
S2. B = 7
```

There’s no anti-dependence.
S2 & S3 can be executed in parallel.

- Output dependence
  - Write-After-Write (WAW)
    - Rewrite with variable renaming



```
S1. B = 3
S2. A = B+1
S3. B = 7
```

```
S1. B2 = 3
S2. A = B2+1
S3. B = 7
```

There’s no output-dependence.
S1 & S3 can be executed in parallel.

Loop Level Parallelism

- Determining whether two statements are dependent is not easy.

- Example

```
1:  a[i] = b[i] + c[i];
2:  d[i] = a[i];
```

  - There is a true dependence,
  - If we put this code in a loop body, the dependence flows within the same iteration.

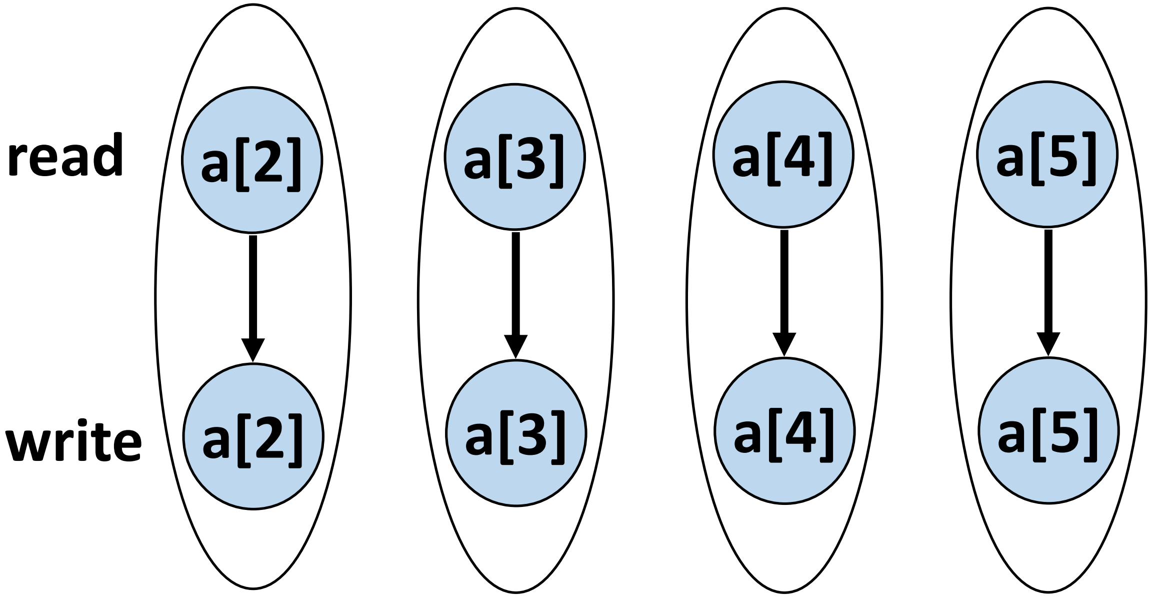
```
for(int i=0;i<N;i++){
    a[i] = b[i] + c[i];      /* 1 */
    d[i] = a[i];             /* 2 */
}
```

  - This dependence is loop-independent.
    - aka.: the dependence distance is 0

Loop Level Parallelism

Example

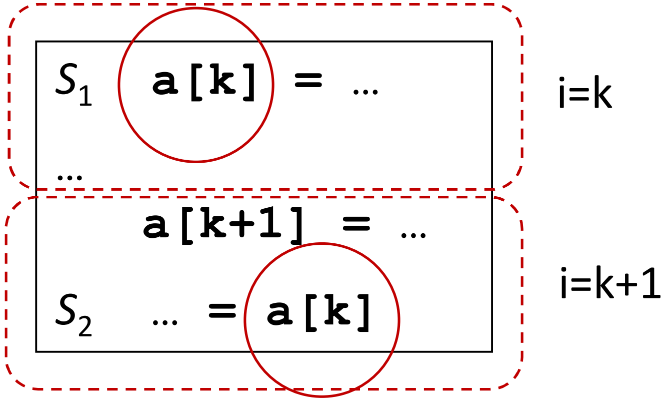
```
for (int i= 2; i<=5; i++){
    a[i] = a[i] + 3
}
```



Loop-Carried Dependence: True Dependence

- Example

```
for(int i=0;i<N;i++){
   a[i] = b[i] + c[i];       /* 1 */
   d[i] = a[i-1];             /* 2 */
}
```



S_1 a[k] = …

…

a[k+1] = …

S_2 … = a[k]

i=k

i=k+1

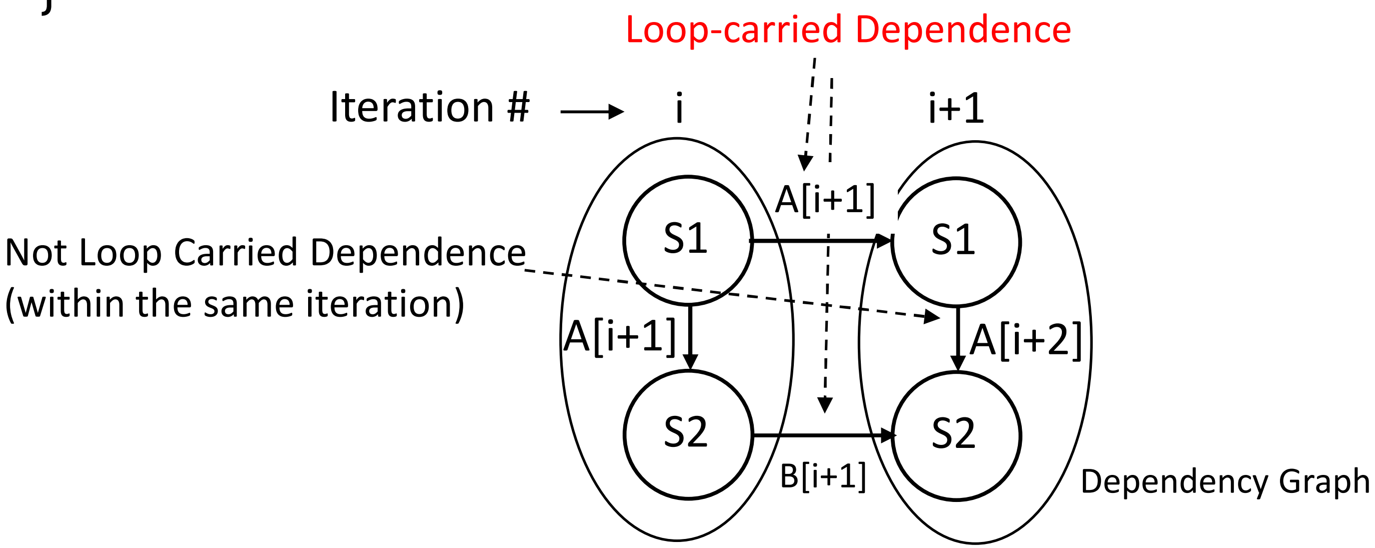
- There is a true dependence
- However, the dependence flows between instances of the statements in different iterations.
  - This is called loop-carried dependence.
  - Aka.: The dependence distance is 1.

  - Note: Loop-carried dependence may prevent parallelism.

Loop-Carried Dependence: True Dependence

Example:

```
for (i=0; i<100; i++)  {
        A[i+1] = A[i] + C[i];      /* S1 */
        B[i+1] = B[i] + A[i+1];  /* S2 */
}
```

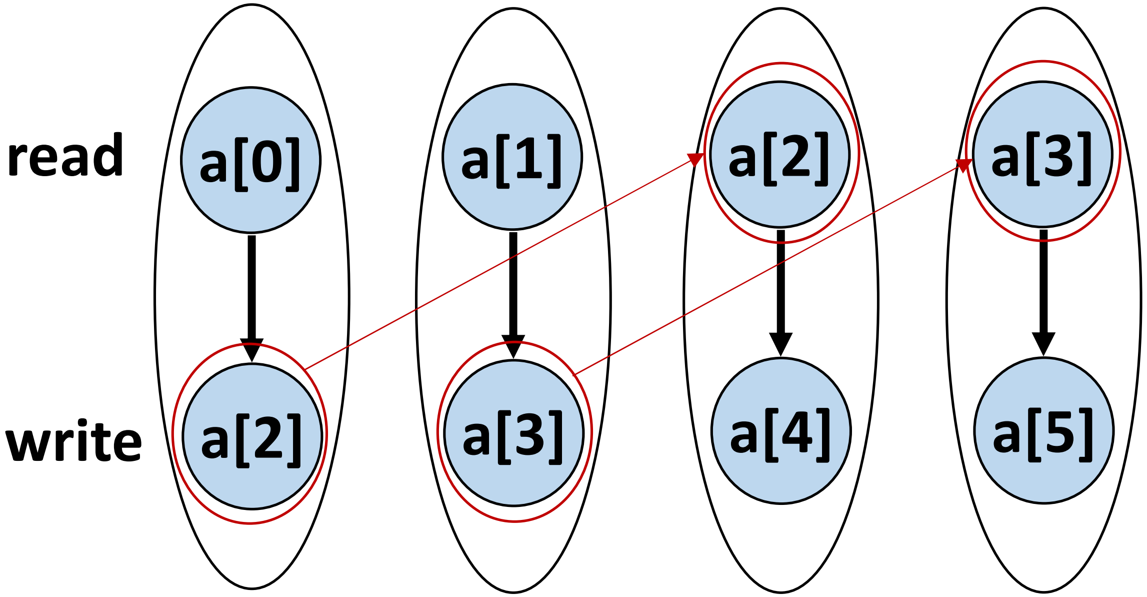


- S1 & S2 use values computed in the earlier iteration
  - These loop-carried dependences prevent loop parallelism.

Loop-Carried Dependence: True Dependence

Example:

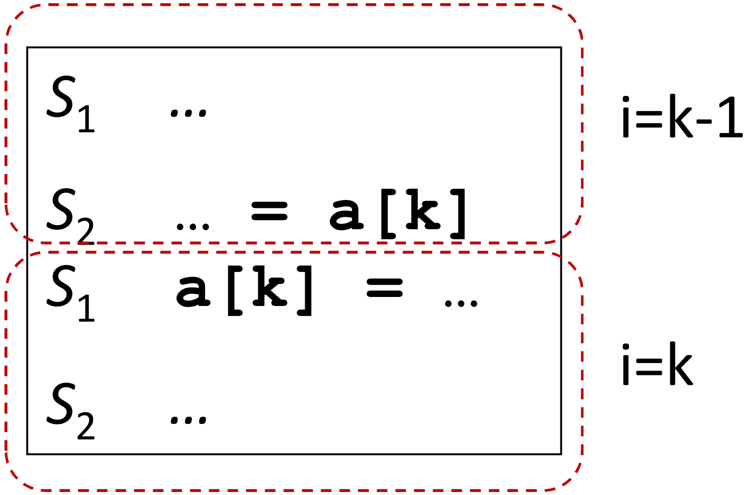
```
    for(int i=2; i<=5; i++){
        a[i] = a[i-2] + 3
    }
```



Loop-Carried Dependence: Anti Dependence

- Example

```
for(int i=0;i<N;i++){
    a[i] = b[i] + c[i];     /* 1 */
    d[i] = a[i+1];          /* 2 */
}
```



S_1 …

S_2 … = `a[k]`

S_1 `a[k]` = …

S_2 …

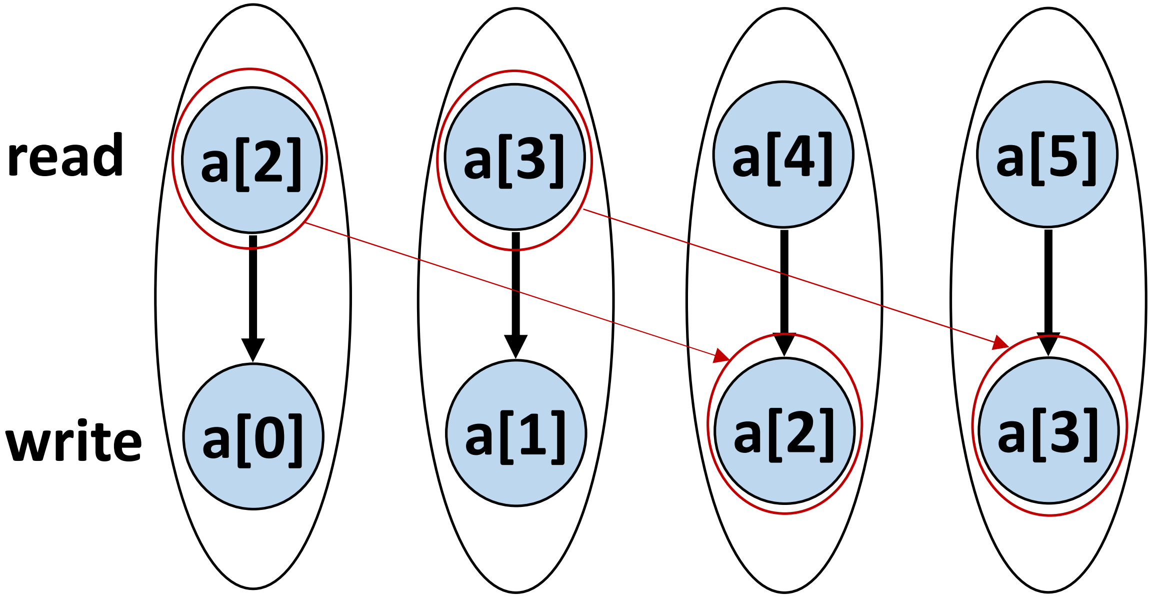
- There is an anti dependence
- This is also a loop-carried dependence.
- The dependence distance is -1.

- Note: Loop-carried dependence may prevent parallelism.

Loop-Carried Dependence: Anti Dependence

Example:

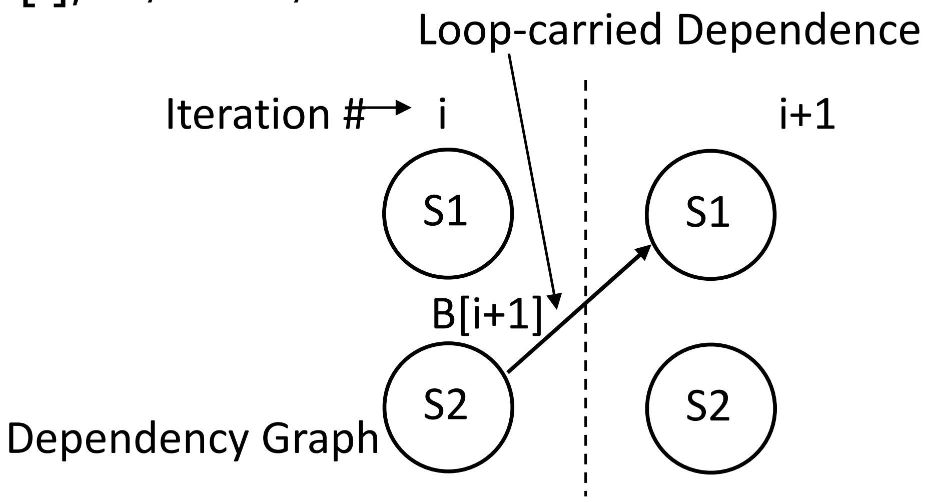
```
for( int i= 2; i<=5; i++){
    a[i-2] = a[i] + 3
}
```



Removing Dependences

- In the loop:

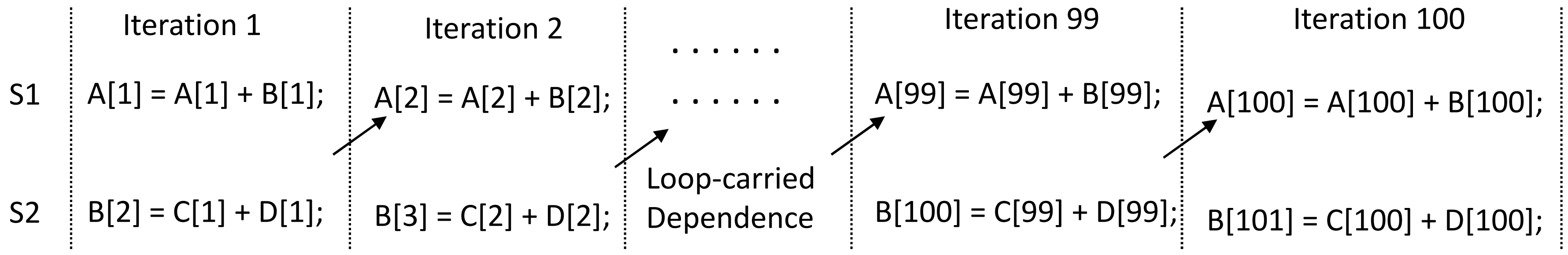
```
for (i=0; i<100; i++)  {
        A[i] = A[i] + B[i];         /* S1 */
        B[i+1] = C[i] + D[i];   /* S2 */
}
```



- S1 uses a value B[i] computed by S2 in the earlier iteration
- This dependence is not circular and does not form a chain.
  - Can be made parallel by replacing the code with …

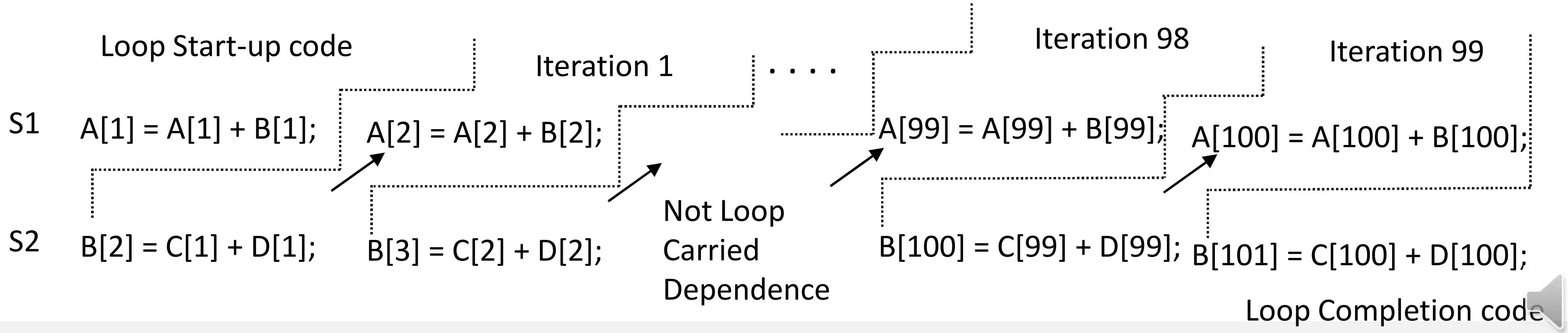
Removing Dependences

Original Loop:



	Iteration 1	Iteration 2	. . . .	Iteration 99	Iteration 100
S1	A[1] = A[1] + B[1];	A[2] = A[2] + B[2];	. . . .	A[99] = A[99] + B[99];	A[100] = A[100] + B[100];
S2	B[2] = C[1] + D[1];	B[3] = C[2] + D[2];	Loop-carried Dependence	B[100] = C[99] + D[99];	B[101] = C[100] + D[100];

How about this?



	Loop Start-up code	Iteration 1	. . . .	Iteration 98	Iteration 99
S1	A[1] = A[1] + B[1];	A[2] = A[2] + B[2];		A[99] = A[99] + B[99];	A[100] = A[100] + B[100];
S2	B[2] = C[1] + D[1];	B[3] = C[2] + D[2];	Not Loop Carried Dependence	B[100] = C[99] + D[99];	B[101] = C[100] + D[100];

Loop Completion code

Removing Dependences

Original Loop:

```
for (i=1; i<100; i++) {
        A[i] = A[i] + B[i];          /*  S1  */
        B[i+1] = C[i] + D[i];     /*  S2  */
}
```

Modified Parallel Loop:

```
A[1] = A[1] + B[1];
for (i=1; i<99; i++)  {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Removing Dependences

- What kind of dependences are there?

```
for (i=0; i<N; i++) {
            x = (B[i] + C[i])/2;
            A[i] = A[i+1] + x;
}
```

- A: Output dependence for x &
  Anti dependence for A[i] = A[i+1]

Removing Dependences

```
#pragma omp parallel for
for (i=0; i<N; i++){
        A2[i] = A[i+1];
}

#pragma omp parallel for private(x)
for (i=0; i<N; i++) {
        x = (B[i] + C[i])/2;
        A[i] = A2[i+1] + x;
}
```

Loop Level Parallelism Analysis

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

Note that S4 has no dependences with other statements

```
for 'i</ : i; n: i**( E
    a@iB< b[i+1] * a[i-1];     /* S1 */
    b@iB< b@iB) coef:          /* S2 */
    c@iB< / -4 ) 'c@iB* a@iB(:  /* S3 */
G
for 'i</ : i; n: i**( E
    d@iB< d[i-1] * d@iB        /* S4 */
G
```

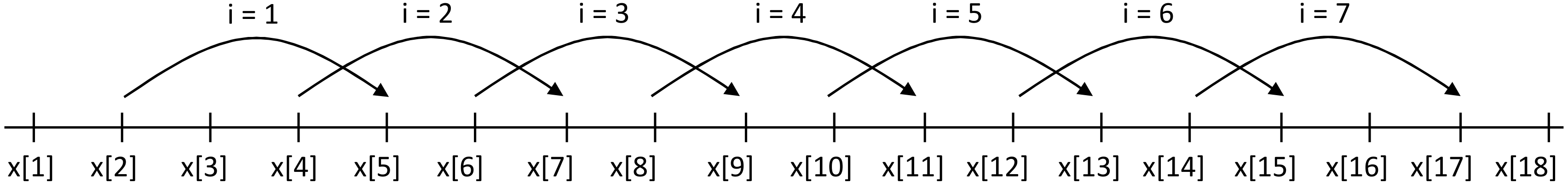
This is called “function parallelism”

Loop Level Parallelism Analysis

- Is there any dependence?

```
for (i=1; i<=100; i=i+1)  {
    A[2*i+3] = A[2*i] + 5;
}
```

- GCD (Greatest Common Divisor) test to detect loop-carried dependence
  - If an array element with index: a*i+b is stored and c*i+d of the same array is loaded later, GCD(c, a) must divide (d-b). I.e., a*i+b = c*i+d
    - E.g.) GCD(a,c) =2, d-b = -3. 2 does not divide -3
    - No loop carried dependence possible
  - GCD test is sufficient to guarantee no loop carried dependence.
  - GCD cannot tell if there `is' a loop carried dependence.

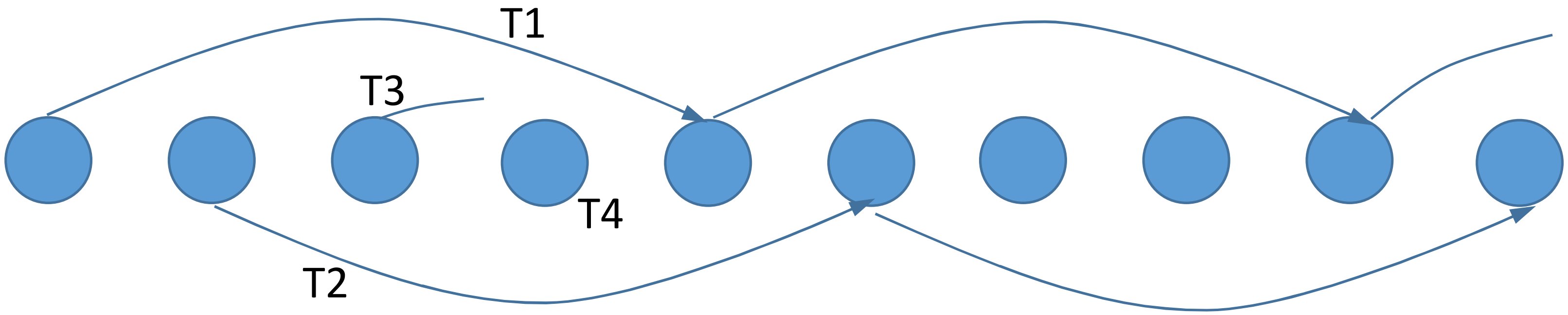


Loop Level Parallelism Analysis

- Is there any dependence?

```
for (i=4; i<104; i++)
        A[i] = 2 * A[i-4];
```

  - Between a[10], a[6], …
  - Between a[11], a[7], …
- Some parallel execution is possible
  - How much?

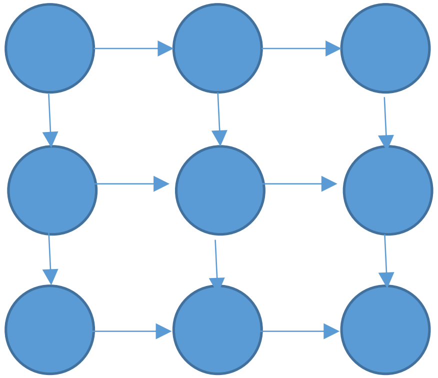


We can divide this loop into four parallel tasks

Loop Level Parallelism Analysis

- Is there any dependence?

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        A[i][j] = A[i][j-1] + A[i-1][j];
```



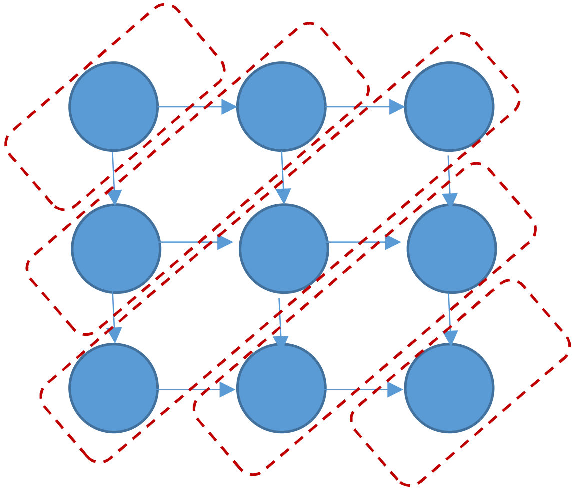
Some parallel execution is possible

How?

Loop Level Parallelism Analysis: Recurrence Pattern

- Is there any dependence?

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        A[i][j] = A[i][j-1] + A[i-1][j];
```



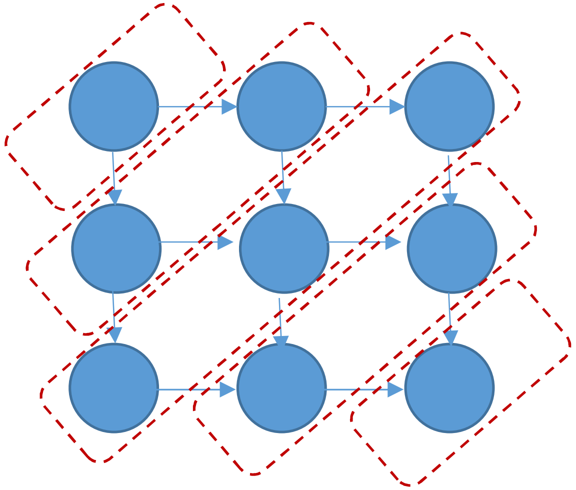
In each diagonal, the nodes are independent of each other

Let’s rewrite the code to iterate over each diagonal

Loop Level Parallelism Analysis: Recurrence Pattern

- Strategy
  - Calculate number of diagonals
  - **for each** diagonal do

    ```
    Calculate the number of points in the current diagonal
    for each point in the current diagonal do
        Compute the value of the current point in the matrix
    ```



Loop Level Parallelism Analysis: Recurrence Pattern

```
for (i=1; i <= 2*n-1; i++) {// 2n-1 anti-diagonals
    if (i <= n) {
        points = i;     // number of points in diag
        row = i;        // first pt (row,col) in diag
        col = 1;
    }
    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++;
    }
}
```