RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS 516 Compilers and Programming Languages II

Parallel Computing-3

Annoucements

Project 1 deadline is Wednesday, March 25 (after spring break)

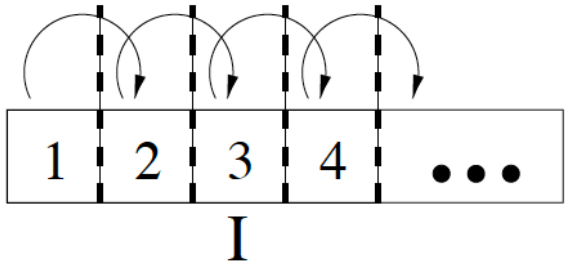
Homework #2 sample solution has been posted

Homework #3 will be posted soon

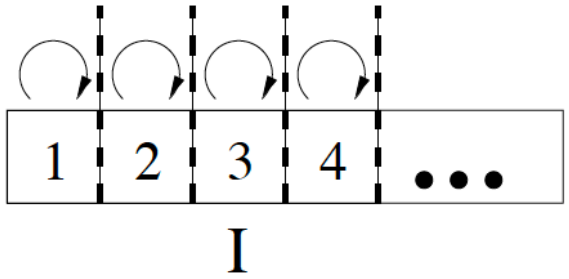
Due to the coronavirus, there is a high chance that we will only conduct our lectures remotely after the spring break, at least for some time. Will be looking into technologies that we will be able to use.

Dependence Analysis for Arrays

```
do I = 1, 100
   A(I) =
      = A(I-1)
enddo
```




```
do I = 1, 100
   A(I) =
      = A(I)
enddo
```



"I" is the iteration space

A loop-independent dependence exists regardless of the loop structure. The source and sink of the dependence occur on the same loop iteration.

A loop-carried dependence is induced by the iterations of a loop. The source and sink of the dependence occur on different loop iterations.

Loop-carried dependences can inhibit parallelization; together with loop-independent dependencies they can limit possible transformations

Iteration Numbers

The iteration number of a loop is equal to the value of the loop index

Definition:

For an arbitrary loop in which the loop index I runs from L to U in steps of S, the iteration number *i* of a specific iteration is equal to the index value I on that iteration

Example:

```
      DO I = 0, 10, 2
S1        <some statement>
      ENDDO
```

Iteration Vectors

What do we do for nested loops?

- Need to consider the nesting level of a loop
- Nesting level of a loop is equal to one more than the number of loops that enclose it.
- Given a nest of n loops, the iteration vector *i* of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level. Thus, the iteration vector is:

$$(i_1, i_2, \ldots, i_n)$$

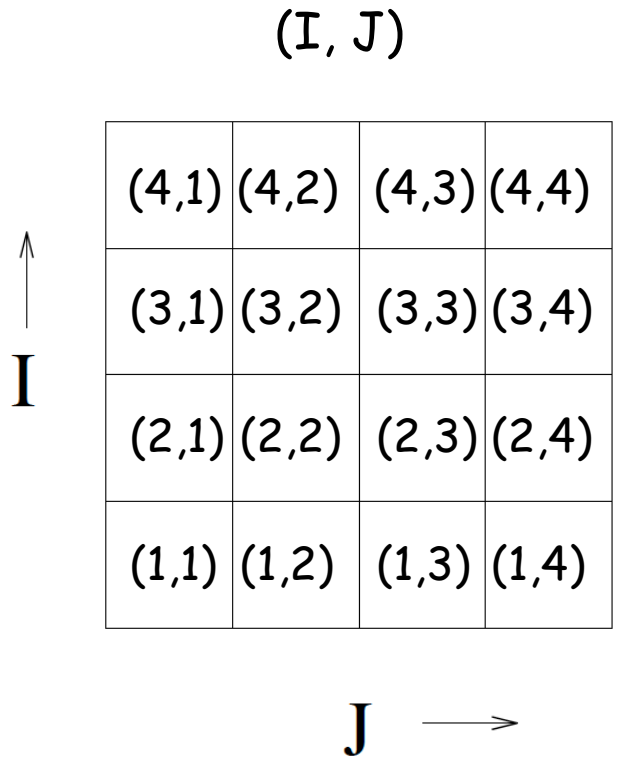
where i_k, $1 \leq k \leq n$ represents the iteration number for the loop at nesting level k

Iteration Vectors

Example:

```
DO I = 1, 4
   DO J = 1, 4
S1        <some statement>
   ENDDO
ENDDO
```

The iteration vector $S_1[(2, 1)]$ denotes the instance of S_1 executed during the 2nd iteration of the I loop and the 1st iteration of the J loop



Ordering of Iteration Vectors

Iteration Space: The set of all possible iteration vectors for a statement

Example:

```
DO I = 1, 2
   DO J = 1, 2
S1        <some statement>
   ENDDO
ENDDO
```

The iteration space for S_1 is { (1,1), (1,2), (2,1), (2,2) }

Ordering of Iteration Vectors

- Useful to define an ordering for iteration vectors
- Define an intuitive, lexicographic order
- Assume two iteration $i = (i_1, \ldots i_n)$ and iteration $j = (j_1, \ldots j_n)$ in a common loop nest; Iteration i precedes iteration j, denoted $i < j$, iff:

$$(i_1, \ldots i_k) = (j_1, \ldots j_k) \text{ and } i_{k+1} < j_{k+1} \quad \text{where } k < n$$

Formal Definition of Loop Dependence

Theorem Loop Dependence:

There exists a dependence from statements S_1 to statement S_2 in a common nest of loops if and only if there exist two iteration vectors *i* and *j* for the nest, such that

(1) $i < j$ or $i = j$ and there is a path from S_1 to S_2 in the body of the loop,
(2) statement S_1 accesses memory location *M* on iteration *i* and statement S_2 accesses location *M* on iteration *j*, and
(3) one of these accesses is a write.

Theorem follows from the definition of dependence

Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program

- But, what is the "meaning" of a program?

For our purposes:

Two computations are equivalent if, on the same inputs, they produce the same outputs in the same order

Reordering Transformations

- A reordering transformation is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statements
- A reordering transformation does not eliminate dependences
- However, it can change the ordering of the dependence which may lead to incorrect behavior
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

Fundamental Theorem of Dependence:

Any reordering transformation that preserves every dependence in a program preserves the meaning of that program, and is called a valid transformation.

Distance and Direction Vectors

- Consider a dependence in a loop nest of n loops
  - → Statement S_1 on iteration i is the source of the dependence
  - → Statement S_2 on iteration j is the sink of the dependence

- The distance vector is a vector of length n, **d(i,j)** , such that:

  $d(i,j) = j - i$, i.e., $d(i,j)_k = j_k - i_k$ (vector subtraction)

- Distance vectors are normalized for loops in which the index step size is not equal to 1.

  From now on, we only look at normalized loops, i.e., step 1

Direction Vectors

Suppose that there is a dependence from statement S_1 on iteration i of a loop nest of n loops and statement S_2 on iteration j, then the direction vector is $D(i,j)$ is defined as a vector of length n such that

$$D(i,j)_k = \begin{cases} \text{"<"} & \text{if } d(i,j)_k > 0 \\ \text{"="} & \text{if } d(i,j)_k = 0 \\ \text{">"} & \text{if } d(i,j)_k < 0 \end{cases}$$

Direction Vectors

- A dependence cannot exist if it has a direction vector whose leftmost non "=" component is not "<" as this would imply that the sink of the dependence occurs before the source.

- Theorem Direction Vector Transformation. Let *T* be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non- "=" component that is ">".

- Follows from Fundamental Theorem of Dependence:
  - → All dependences exist
  - → None of the dependences have been reversed

Distance/Direction Vectors

Example:

Consistent dependencies:
Pairs of dependent iterations have the same distance

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
S1       A(I+1, J, K-1) = A(I, J, K) + 10
      ENDDO
   ENDDO
ENDDO
```

S_1 has a true dependence on itself.

Distance Vector: (1, 0, -1)

Direction Vector: (<, =, >)

Distance/Direction Vectors

Example:

Consistent dependencies:
Pairs of dependent iterations have the same distance

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
S1       A(I+1, J, K-1) = A(I, J, K) + 10
      ENDDO
   ENDDO
ENDDO
```

What loop interchange transformation is valid?

Swap I and J

Swap J and K

Swap I and K

S_1 has a true dependence on itself.

Distance Vector: (1, 0, -1)

Direction Vector: (<, =, >)

Distance/Direction Vectors

Example:

Consistent dependencies:
Pairs of dependent iterations have the same distance

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
S1       A(I+1, J, K-1) = A(I, J, K) + 10
      ENDDO
   ENDDO
ENDDO
```

S_1 has a true dependence on itself.

Distance Vector: (1, 0, -1)

Direction Vector: (<, =, >)

What loop interchange transformation is valid?

Swap I and J
(0, 1, -1) and (=, <, >)

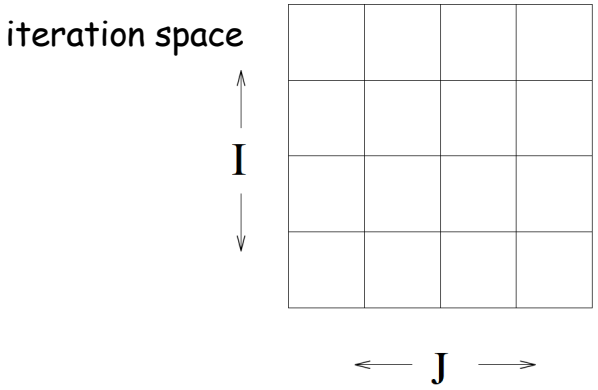
Swap J and K
(1, -1, 0) and (<, >, =)

Swap I and K
(-1, 0, 1) and (>, =, <)
not valid

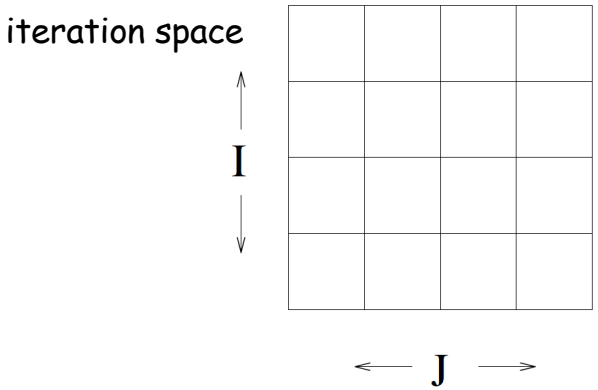
Distance/Direction Vectors

More examples

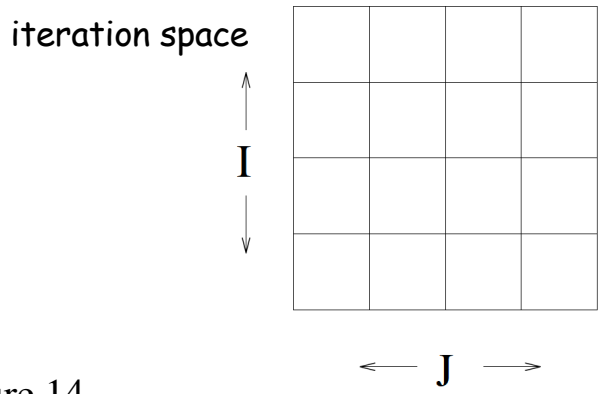
```
DO I = 1, N
   DO J = 1, N
S1        A(I,J) = A(I,J-1)
   ENDDO
ENDDO
```




```
DO I = 1, N
   DO J = 1, N
S1        A(I,J) = A(I-1,J-1)
   ENDDO
ENDDO
```



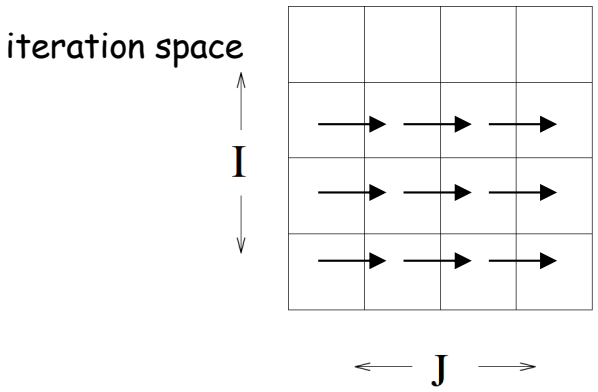

```
DO I = 1, N
   DO J = 1, N
S1        A(I,J) = A(I-1,J+1)
   ENDDO
ENDDO
```



Distance/Direction Vectors

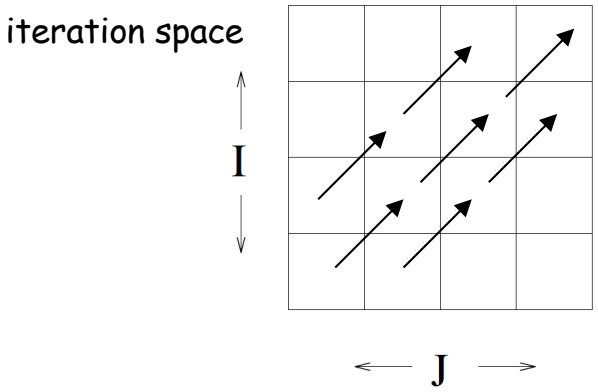
More examples

```
DO I = 1, N
   DO J = 1, N
S1        A(I,J) = A(I,J-1)
   ENDDO
ENDDO
```



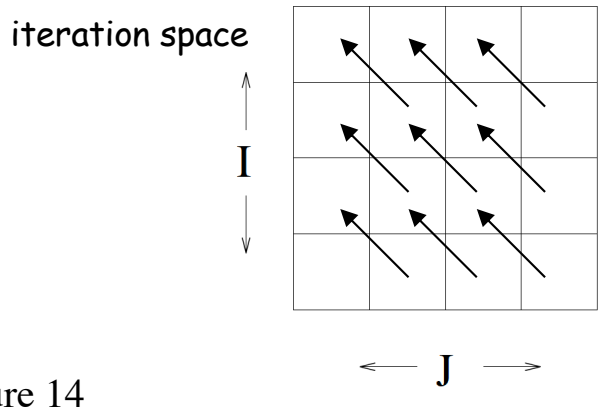
(0, 1)
(=, <)

```
DO I = 1, N
   DO J = 1, N
S1        A(I,J) = A(I-1,J-1)
   ENDDO
ENDDO
```



(1, 1)
(<, <)

```
DO I = 1, N
   DO J = 1, N
S1        A(I,J) = A(I-1,J+1)
   ENDDO
ENDDO
```



(1,-1)
(<, >)

Loop-carried and Loop-independent Dependences

- If in a loop statement S_2 depends on S_1, then there are two possible ways of this dependence occurring:

1. S_1 and S_2 execute on different iterations
   → This is called a loop-carried dependence.

2. S_1 and S_2 execute on the same iteration
   → This is called a loop-independent dependence.

Loop-carried dependence

Definition:

Statement S_2 has a loop-carried dependence on statement S_1 if and only if S_1 references location M on iteration i, S_2 references M on iteration j and $d(i,j) > 0$ (that is, $D(i,j)$ contains a "<" as leftmost non "=" component).

Example:

```
DO I = 1, N
S1      A(I+1) = F(I)
S2      F(I+1) = A(I)
ENDDO
```

Loop-carried dependence

Level of a loop-carried dependence is the index of the leftmost non-"=" of D(i,j) for the dependence.

For instance:

```
    DO I = 1, 10
       DO J = 1, 10
          DO K = 1, 10
S1          A(I, J, K+1) = A(I, J, K)
          ENDDO
       ENDDO
    ENDDO
```

- Direction vector for S1 is (=, =, <)
- Level of the dependence is 3
- A level-k dependence between S_1 and S_2 is denoted by $S_1 \delta_k S_2$

Loop-carried Transformations

- Theorem Any reordering transformation that does not alter the relative order of any loops in the nest and preserves the iteration order of the level-*k* loop preserves all level-*k* dependences.
- Proof:
  - → D(i, j) has a "<" in the k^{th} position and "=" in positions 1 through k-1
  - ⇒ Source and sink of dependence are in the same iteration of loops 1 through k-1
  - ⇒ Cannot change the sense of the dependence by a reordering of iterations of those loops
- As a result of the theorem, powerful transformations can be applied