Loop Parallelization Techniques

- Data-Dependence Analysis
- Dependence-Removing Techniques
- Parallelizing Transformations
- Performance-enhancing Techniques

Some motivating examples

```
Do i = 1, n
   a(i) = b(i)      S1
   c(i) = a(i-1)    S2
End do
```

Is it legal to

- Run the i loop in parallel?
- Put S_2 first in the loop?

```
Do I = 1, n
  a(i) = b(i)
End do

Do I = 1, n
  c(i) = a(i-1)
End do
```

Is it legal to

- Fuse the two i loops?

In general, it is desirable to determine if two references access the same memory location, and the order they execute, so that we can determine if the references might execute in a different order after some transformation.

Dependence, an example

```
Do i = 1, n
  a(i) = b(i)      S1
  c(i) = a(i-1)    S2
End do
```

→ Indicates dependences, i.e. the statement at the head of the arc is somehow dependent on the statement at the tail

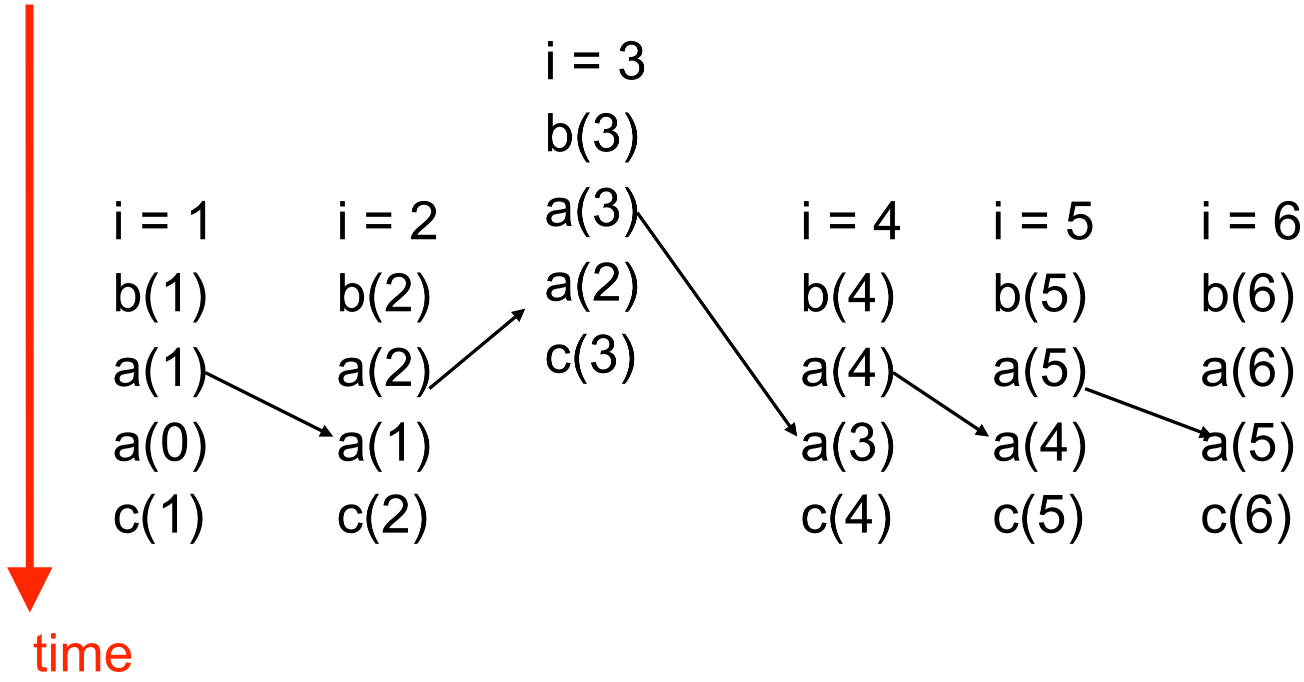
i = 1	i = 2	i = 3	i = 4	i = 5	i = 6
b(1)	b(2)	b(3)	b(4)	b(5)	b(6)
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)
a(0)	a(1)	a(2)	a(3)	a(4)	a(5)
c(1)	c(2)	c(3)	c(4)	c(5)	c(6)

Can this loop be run in parallel?

```
Do i = 1, n
  a(i) = b(i)     S1
  c(i) = a(i-1)   S2
End do
```

Assume 1 iteration per processor, then if for some reason some iterations execute out of lock-step, bad things can happen

In this case, read of a(2) in i=3 will get an invalid value!



Can we change the order of the statements if the loop is serial?

```
Do i = 1, n
   a(i) = b(i)      S1
   c(i) = a(i-1)    S2
End do
```

```
Do i = 1, n
   c(i) = a(i-1)    S2
   a(i) = b(i)      S1
End do
```

No problem with a serial execution.

Access order before statement reordering

b(1) a(1) a(0) c(1) || b(2) a(2) a(1) c(2) || b(3) a(3) a(2) c(3) || b(4) a(4) a(3) c(4)

i=1 *i=2* *i=3* *i=4*

Access order after statement reordering

a(0) c(1) b(1) a(1) || a(1) c(2) b(2) a(2) || a(2) c(3) b(3) a(3) || a(3) c(4) b(4) a(4)

i=1 *i=2* *i=3* *i=4*

Types of dependence

$$a(2) = \ldots \xrightarrow{\delta^f} \ldots = a(2)$$

Flow or true dependence – data for a read comes from a previous write (write/read hazard in hardware terms_

$$\ldots = a(2) \xrightarrow{\delta^a} a(2) = \ldots$$

Anti-dependence – write to a location cannot occur before a previous read is finished

$$a(2) = \ldots \xrightarrow{\delta^o} a(2) = \ldots$$

Output dependence – write a location must wait for a previous write to finish

Dependences always go from earlier in a program execution to later in the execution

Anti and output dependences can be eliminated by using more storage.

Eliminating anti-dependence

… = a(2)
a(2) = …

Anti-dependence – write to a location cannot occur before a previous read is finished

Let the program in be:

a(2) = …
… = a(2)
a(2) = …
= … a(2)

Create additional storage to eliminate the anti-dependence

The new program is:

a(2) = …
… = a(2)
aa(2) = …
= … aa(2)

No more anti-dependence!

Getting rid of output dependences

a(2) = …

a(2) = …

Output dependence – write a location must wait for a previous write to finish

Let the program be:

a(2) = …

… = a(2)

a(2) = …

… = a(2)

Again, by creating new storage we can eliminate the output dependence.

The new program is:

a(2) = …

… = a(2)

aa(2) = …

… = aa(2)

Eliminating dependences

- In theory, can always get rid of anti- and output dependences
- Only flow dependences are inherent, i.e. must exist, thus the name “true” dependence.
- In practice, it can be complicated to figure out how to create the new storage
- Storage is not free – cost of creating new variables may be greater than the benefit of eliminating the dependence.

Can we fuse the loop?

```
Do i = 1, n
   a(i) = b(i)      S1
End do
Do i
   c(i) = a(i-1)    S2
End do
```

```
Do i = 1, n
   a(i) = b(i)      S1
   c(i) = a(i-1)    S2
End do
```

In original execution of the unfused loops:

1. A(i-1) gets value assigned in a(i)
2. Can't overwrite value assigned to a(i) or c(i)
3. B(i) value comes from outside the loop

1. Is ok after fusing, because get a(i-1) from the value assigned in the previous iteration
2. No "output" dependence on a(i) or c(i), not overwritten
3. No input flow, or true dependence on a b(i), so value comes from outside of the loop nest

Data Dependence Tests:
Other Motivating Examples

Statement Reordering
can these two statements be swapped?

```
DO i=1,100,2
   B(2*i) = ...
      ...    = B(3*i)
ENDDO
```

Loop Parallelization
Can the iterations of this loop be run concurrently?

```
DO i=1,100,2
   B(2*i) = ...
      ...    = B(2*i) +B(3*i)
ENDDO
```

An array data dependence exists between two data references iff:

- both references access the same storage location
- at least one of them is a write access

Dependence sources and sinks

- The *sink* of a dependence is the statement at the head of the dependence arrow
- The *source* is the statement at the tail of the dependence arrow

```
for (i=1; i < nl i++) {
   a[i] = …

    … = a[i-1]
}

a[1] =
       = a[0]
a[2] =
       = a[1]
a[3] =
       = a[2]
a[4] =
       = a[3]
```

Data Dependence Tests: Concepts

Terms for data dependences between statements of loop iterations.

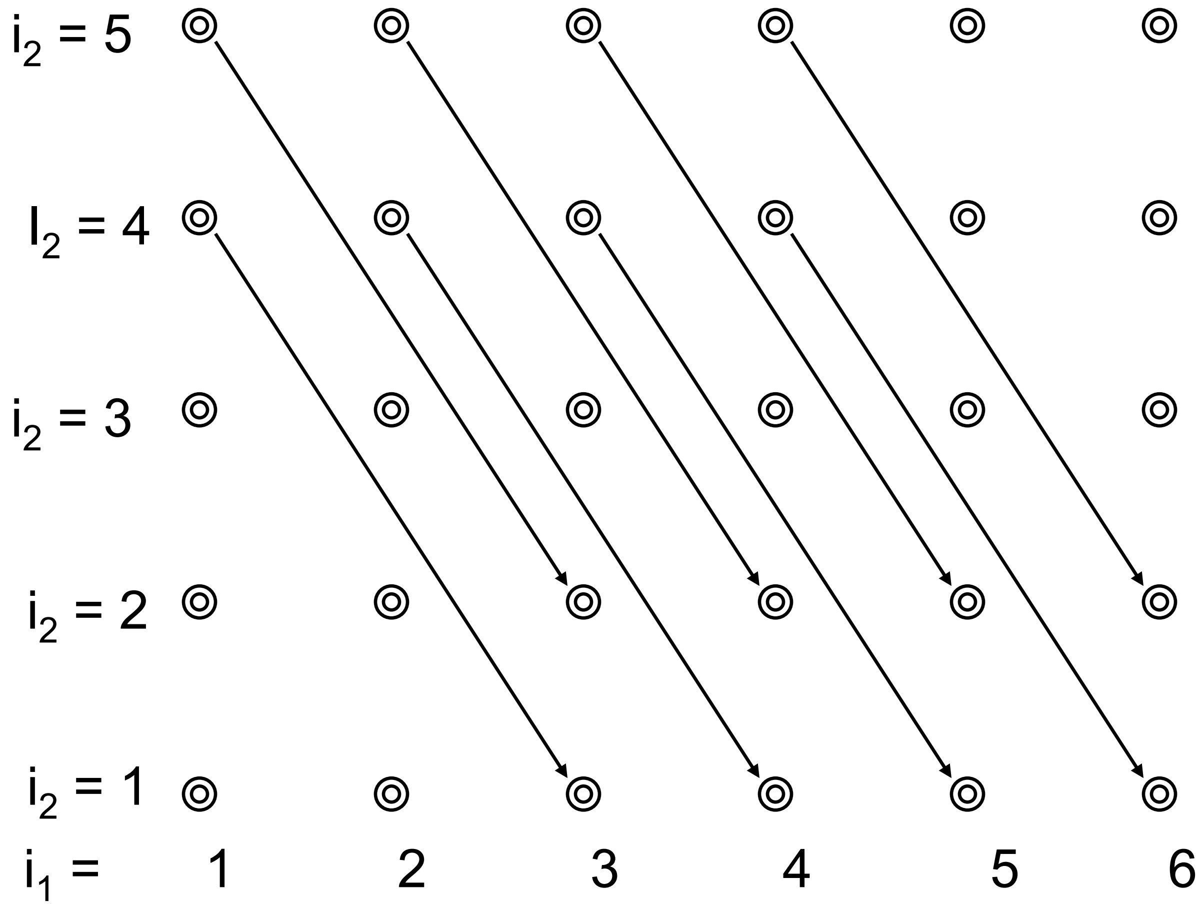
- Distance (vector): indicates how many iterations apart are the source and sink of dependence.
- Direction (vector): is basically the sign of the distance. There are different notations: (<,=,>) or (+1,0,-1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.
- Loop-carried (or cross-iteration) dependence and non-loop-carried (or loop-independent) dependence: indicates whether or not a dependence exists within one iteration or across iterations.
  – For detecting parallel loops, only cross-iteration dependences matter.
  – *equal* dependences are relevant for optimizations such as statement reordering and loop distribution.
- Iteration space graphs: the un-abstracted form of a dependence graph with one node per statement instance.

Data Dependence Tests: Iteration space graphs

- Iteration space graphs: the un-abstracted form of a dependence graph with one node per statement instance.

```
do i1 = 1, n
  do i2 = 1, n
    a(i1,i2) = a(i1-2,i2+3)
  end do
end do
```

This is an iteration space graph (or diagram) →



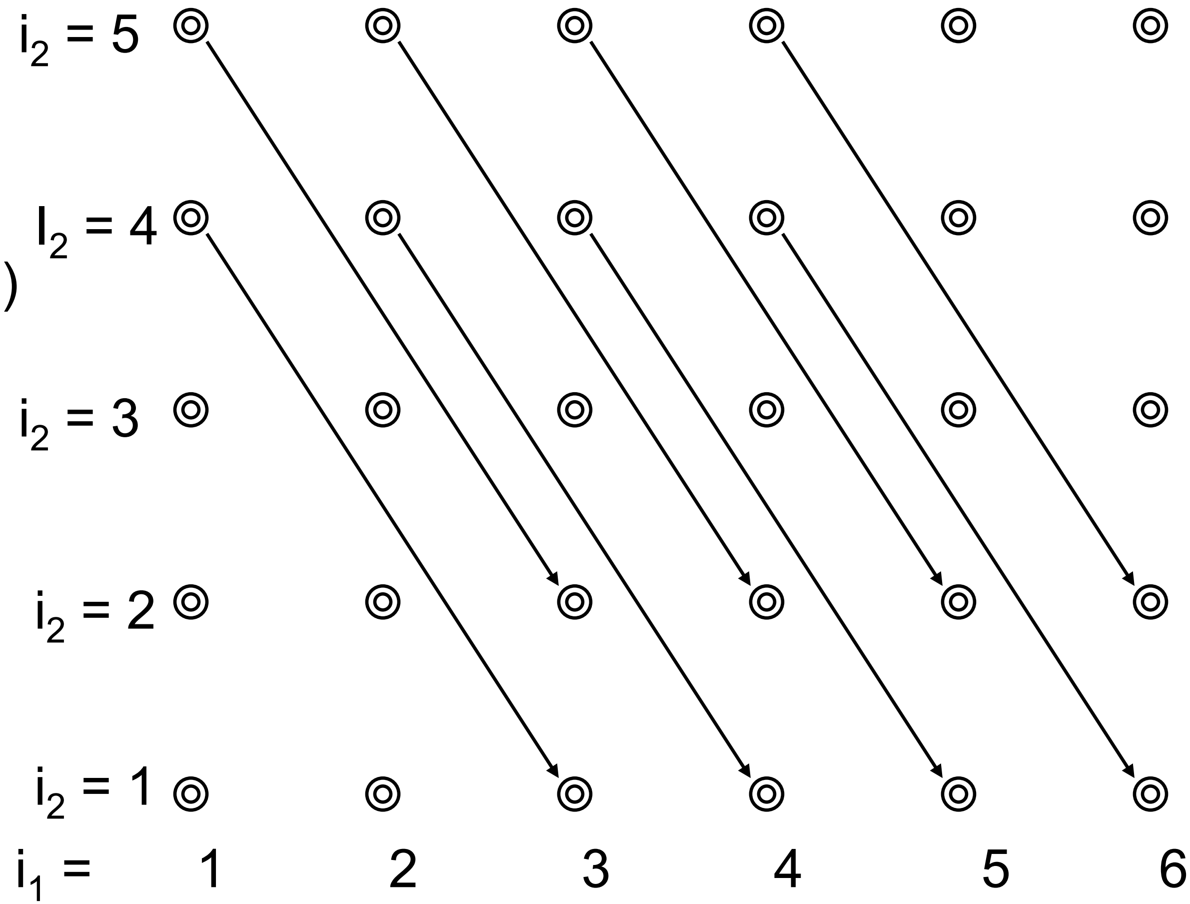
Data Dependence Tests: Distance Vectors

Distance (vector): indicates how many iterations apart are the source and sink of dependence.

```
do i1 = 1, n
  do i2 = 1, n
    a(i1,i2) = a(i1-2,i2+3)
  end do
end do
```

$I=(1,4) \rightarrow I' = (3,1)$

$I' - I = (2,-3)$



Data Dependence Tests: Direction Vectors

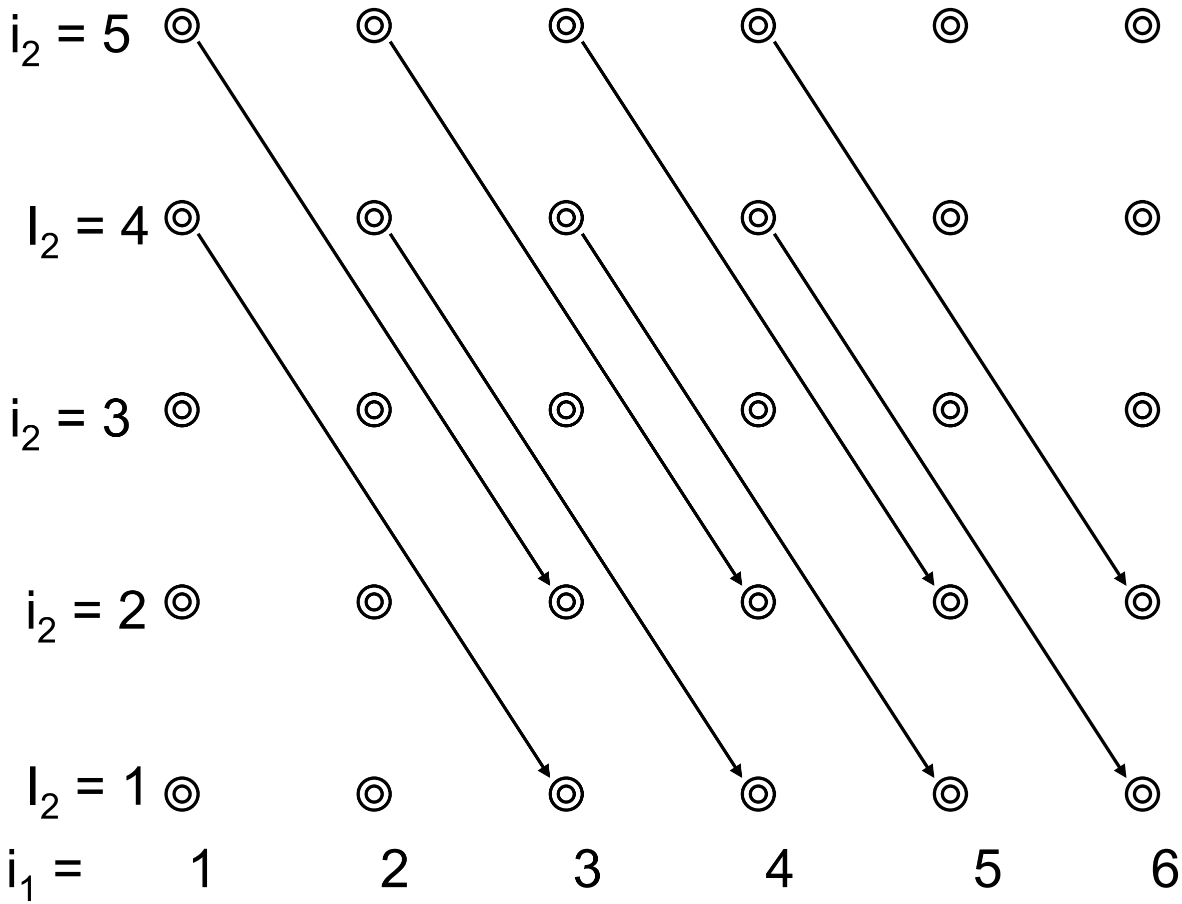
Direction (vector): is basically the sign of the distance. There are different notations: (<,=,>) or (-1,0,+1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.

```
do i1 = 1, n
  do i2 = 1, n
    a(i1,i2) = a(i1-2,i2+3)
  end do
end do
```

$I=(1,4) \rightarrow I' = (3,1)$

$I' - I = d = (2,-3)$

Direction = (<,>) (or (sign(2),sign(-3) == (+1,-1) in some papers



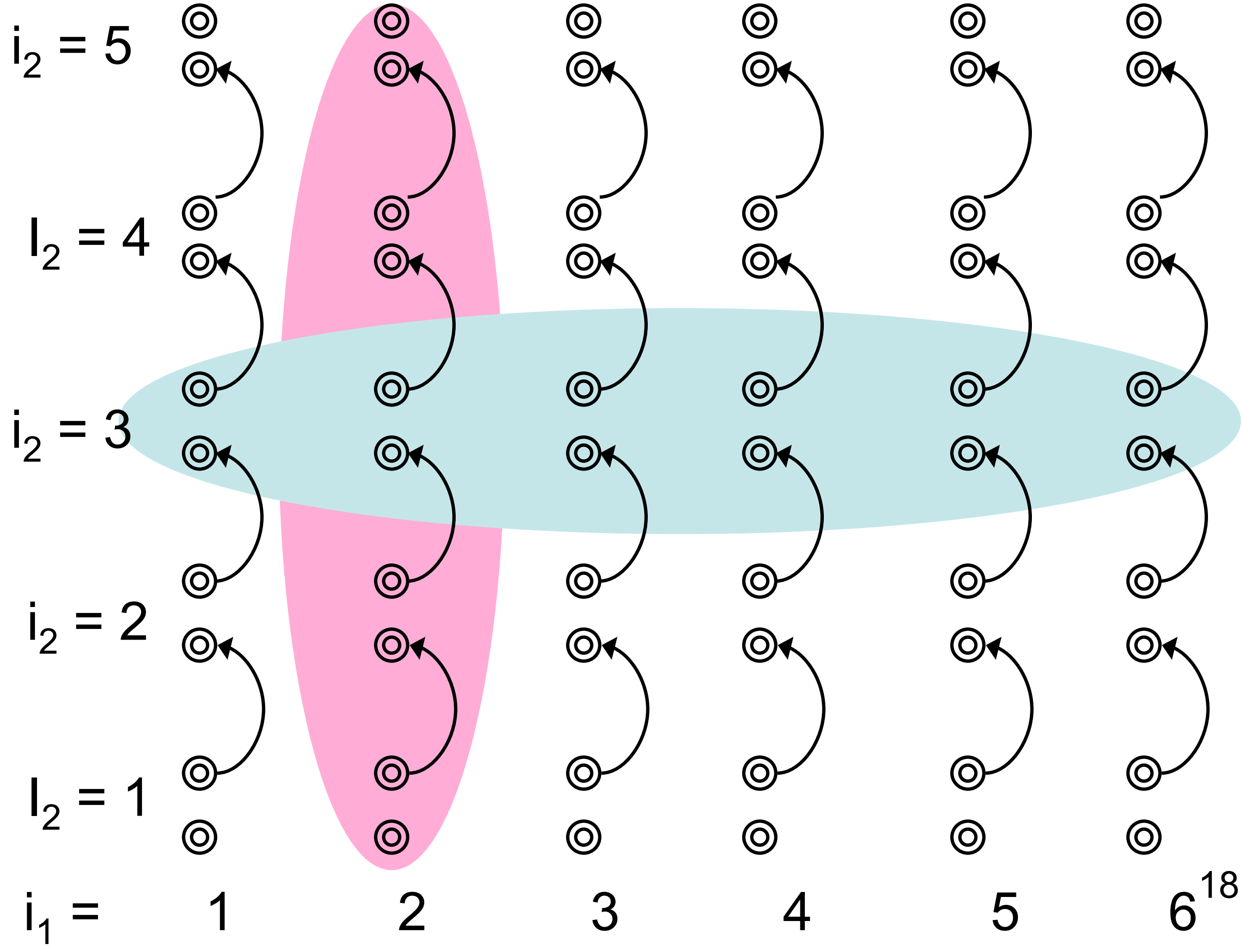
Data Dependence Tests: Loop Carried

- Loop-carried (or cross-iteration) dependence and non-loop-carried (or loop-independent) dependence: indicates whether or not a dependence exists within one iteration or across iterations.

```
do i1 = 1, n
   do i2 = 1, n
      a(i1,i2) =
               = a(i1,i2-1)
   end do
end do
```

Dependence on the i_2 loop is loop carried.

Dependence on the i_1 loop is not.



A quick aside

A loop

```
do i = 4, n, 3
  a(i)
end do
```

Can be always be *normalized* to the loop ➔

```
do i = 0, (n-1)/3-1, 1
  a(3*i+4)
end do
```

This makes discussing the data-dependence problem easier since we only worry about loops from 1, n, 1

More precisely, do i = lower, upper, stride { a(i)} becomes
do i’ = 0, (upper – lower + incre)/stride – 1, 1 {a(i’*stride + lower)}

Data Dependence Tests:
Formulation of the Data-dependence problem

```
DO i=1,n
  a(4*i) = . . .
  . . .    =  a(2*i+1)
ENDDO
```

the question to answer:
can 4*i ever be equal to 2*i'+1 where i, i' ∈[1,n]?
If so, what is the relation of i and i' when they are equal?

In general, given:
- two subscript functions *f(i)* and *g(i')* and
- loop bounds lower, upper.

Does
f(i) = *g(i')* have an ***integer*** solution such that
lower ≤ *i, i'* ≤ *upper*?

Diophantine equations

- An equation whose coefficients and solutions are all integers is a Diophantine equation
- Determining if a Diophantine equation has a solution requires a slight detour into elementary number theory
- Let f(i) = a*i + c and g(i') = b*i' + c', then
  - f(i) = g(i') $\Rightarrow$ a*i - b*i' = c' - c
  - fits general form of linear or *affine* Diophantine equation of $\boldsymbol{a_1 * i_1 + a_2 * i_2 = c}$

Does f(i) = g(i) have a solution?

- The Diophantine equation

$$a_1 * i_1 + a_2 * i_2 = c$$

has a solution *iff* $\gcd(a_1, a_2)$ evenly divides c

Examples:

15*i +6*j -9*k = 12	has a solution	gcd=3
2*i + 7*j = 3	has a solution	gcd=1
9*i + 3*j + 6*k = 5	has no solution	gcd=3

Euclid Algorithm: find gcd(a,b)

```
Repeat
   a ← a mod b
   swap a,b
Until b=0
```

→ *The resulting a is the gcd*

for more than two numbers:
gcd(a,b,c) = (gcd(a,gcd(b,c))

Why $gcd(a_1,a_2)$ *evenly divides c* implies equation has a solution

Let $g = \gcd(a_1,a_2)$, can rewrite the equation as:

$$g*a'_1*i_1 + g*a'_2*i_2 = c \Rightarrow g*(a'_1*i_1+a'_2*i_2) = c$$

Because a'_1 and a'_2 are relatively prime, all integers can be expressed as a *linear combination* of a'_1 and a'_2.

$a'_1*i_1+a'_2*i_2$ is just such a linear combination and therefore $a'_1*i_1–a'_2*i_2$ generates all integers $i \in I$, (assuming i_1, i_2 can range over the integers.)

If remainder(c/g) = 0, c is a solution since c = g*c', and $g*(a'_1*i_1–a'_2*i_2)$ generates all multiples of g.

If remainder(c/g) != 0, c cannot be a solution, since all values generated by $g*(a'_1*i_1–a'_2*i_2)$ are (trivially) divisible by g, and cannot equal any c that is not divisible by g.

More information on gcd's and dependence analysis

- General books on number theory
- Books by Utpal Banerjee (Kluwer Academic Publishers), (Illinois, now Intel) who developed the GCD test in late 70's, Mike Wolfe, (Illinois, now Portland Group) "High Performance Compilers for Parallel Computing
- Randy Allen's thesis, Rice University
- Work by Eigenman & Blume Purdue (range test)
- Work by Pugh (Omega test) Maryland
- Work by Hoeflinger, etc. Illinois (LMAD)

Other Data Dependence Tests

- The GCD test is simple but not very useful
  - Most subscript coefficients are 1, gcd(1,i) = 1
- Other tests
  - ***Banerjee-wolfe test***: accurate state-of-the-art test, takes direction and loop bounds into account
  - ***Omega test***: "precise" test, most accurate for linear subscripts (See Bill Pugh publications for details). Worst case complexity is bad.
  - ***Range test***: handles non-linear and symbolic subscripts (Blume and Eigenmann)
  - many variants of these tests
- Compilers tend to perform simple to complex tests in an attempt to disprove dependence

What do dependence tests do?

- Some tests, and Banerjee's in some situations (affine subscripts, rectangular loops) are precise
  - Definitively proves existence or lack of a dependence
- Most of the time tests are conservative
  - Always indicate a dependence if one may exist
  - May indicate a dependence if it does not exist
- In the case of "may" dependence, run-time test or speculation can prove or disprove the existence of a dependence
- Short answer: tests disprove dependences for some dependences

Banerjee's Inequalities

If $a*i_1 - b*i'_1 = c$ has a solution, does it have a solution within the loop bounds, and for a given direction vector?

```
do i = 1, 100
   x(i) =
        = x(i-1)
end do
```

Note: there is a (<) dependence.

Let's test for (=) and (<) dependence.

By the mean value theorem, c can be a solution to the equation $f(i) = c, i \in [lb,ub]$ iff

- $f(lb) <= c$
- $f(ub) >= c$

(assumes *f(i)* is monotonically increasing over the range *[lb,ub]). Linearity of* f *insures monotonicity,switch* ub*,* lb *if not increasing.*

The idea behind ***Banerjee's Inequalities*** is to find the maximum and minimum values the dependence equation can take on for a given direction vector, and see if these bound *c*. ***This is done in the real domain since integer solution requires integer programming (in NP)***

Example of where the direction vector makes a difference

```
do i = 1, 100
  x(i) =
        = x(i-1)
end do
```

Note: there is a (<) dependence.

Let's test for (=) and (<) dependence.

Dependence equation is **i-i' = -1**

If i = i', then **i-i' = 0**, $\forall$ i, i'

If i < i', then i-i' $\neq$ 0, and when **i'=i+1**, the equation has a solution.

Banerjee test

If $a*i_1 - b*i'_1 = c$ has a solution, does it have a solution within the loop bounds for a given direction vector (<) or (=) in this case)?

For our problem, does $i_1 – i'_1 = -1$ have a solution

- For $i_1 = i'_1$, then it does not (no (=) dependence).
- For $i_1 < i'_1$, then it does ((<) dependence).

When can a loop be made parallel?

- When it has no loop carried dependences
- Distribution can be used to find more parallel loops
- Dependence-like analysis can be used to find data that needs to be sent as messages

No loop carried dependences

```
for (i=0; i < n; i++) {
   a(i) = . . .
        = a(i) + . . .
}
```

a(0) = → = a(0)

a(1) = → = a(1)

a(2) = → = a(2)

. . .

a(n-1) = → = a(n-1)

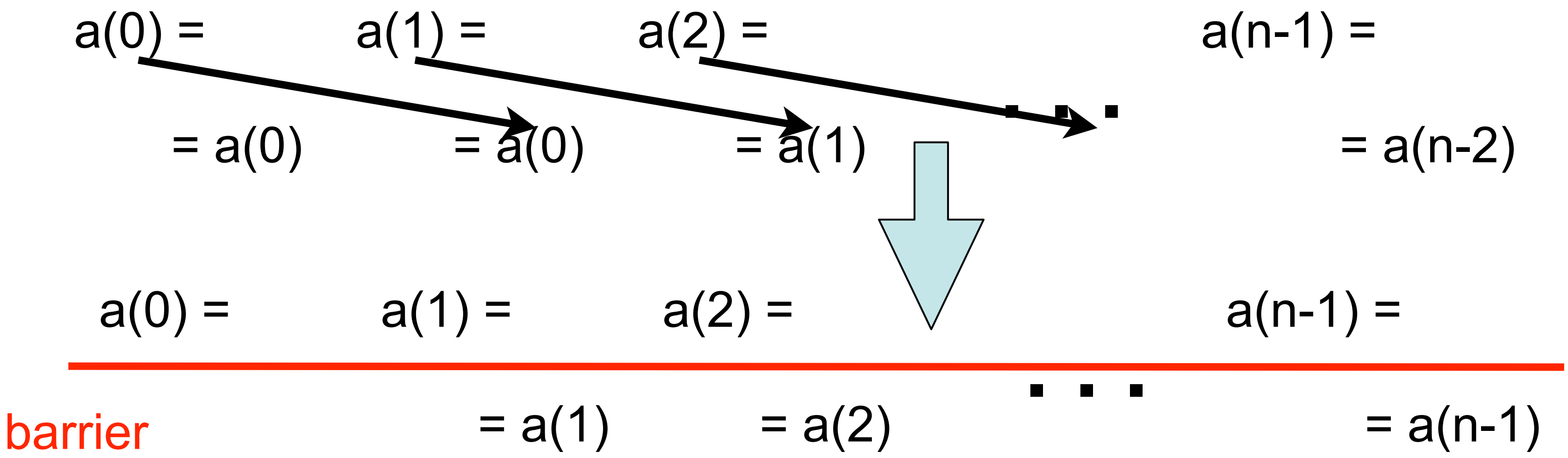
Each iteration is independent and the loop can be executed in parallel

The usefulness of distribution

```
for (i=0; i < n; i++) {
   a(i) = . . .
        = a(i-1) . . .
}
```

```
for (i=0; i < n; i++)
   a(i) = . . .

for (i=0; i < n; i++)
        = a(i-1) . . .
```



What messages must be sent?

Let N=100, block distribution over two processes with elements 0:49 on process P0, 50:99 on P2

```
for (i=0; i < n; i++)
   a(i) = . . .
```

P0 writes a(0:49)
P1 writes a(50:99)

```
for (i=0; i < n; i++)
        = a(i-1) . . .
```

P0 reads a(-1:48)
P1 reads a(49:98)

P0 sends P1 [0,1,2, ..., 49] $\cap$ [49,1,2, ..., 98]

This is the solution of the diophantine equation

$i_1 = i_2-1, \;\; 0 \le i_1 \le 49, \; 50 \le i_1 \le 99$

Loop Fusion and Distribution

```
DO j = 1,n
   a(j) = b(j)
ENDDO

DO k=1,n
  c(k) = a(k)
ENDDO
```

fusion →

← distribution

```
DO j = 1,n
   a(j) = b(j)
   c(j) = a(j)
ENDDO
```

- necessary form for vectorization
- can provide synchronization necessary for “forward” dependences
- can create perfectly nested loops

- less parallel loop startup overhead
- can increase *affinity* (better locality of reference)

Both transformations change the statement execution order. Data dependences need to be considered!

Loop Fusion and Distribution

```
DO j = 1,n
  a(j) = b(j)
ENDDO

DO k=1,n
  c(k) = a(k)
ENDDO
```

fusion →

← distribution

```
DO j = 1,n
  a(j) = b(j)
  c(j) = a(j)
ENDDO
```

Dependence analysis needed:

- Determine uses/def and def/use chains across unfused loops
- Every def ⇒use link should have a flow dependence in the fused loop
- Every use ⇒def link should have an anti-dependence in the fused loop
- No dependence not associated with a use ⇒def or def ⇒use should be present in the fused loop

Loop Fusion and Distribution

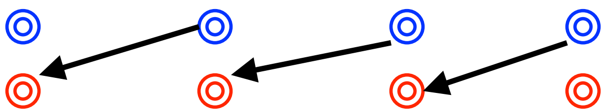
```
DO j = 1,n
   S1: a(j-1) = b(j)
ENDDO

DO k=1,n
  S2: c(k) = a(k)
ENDDO
```

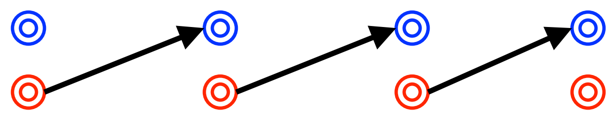
fusion →

← distribution

```
DO j = 1,n
   S1: a(j-1) = b(j)
   S2: c(j) = a(j)
ENDDO
```



J, k= 1   2   3   4



J = 1   2   3   4

Inter-loop “flow: dependence from S1 to S2

Cross iteration anti-dependence from S2 to S1

Dependence graphs

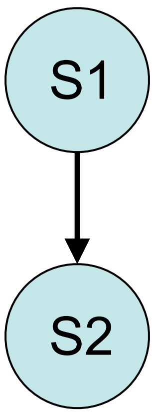
```
DO j = 1,n
  S1: a(j-1) = b(j)
ENDDO

DO k=1,n
  S2: c(k) = a(k)
ENDDO
```

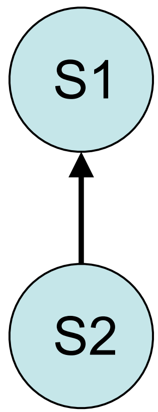
fusion →

← distribution

```
DO j = 1,n
  S1: a(j-1) = b(j)
  S2: c(j) = a(j)
ENDDO
```



δ^f



$\delta^a(<)$

Criteria for Parallelization

- Vectorization:
  - no “lexically backward dependence”.
  - If we allow statement reordering: no dependence cycles
- Parallelization:
  - no loop-carried dependence

    Note, loops inside a dependence-carrying loop are dependence free (w.r.t. a given reference pair)

Dependences

- Preclude parallelization on the loop that has loop carried dependences unless they can be eliminated.
- In shared memory programs, dependences result in loads and stores to the same memory location by different tasks
- In distributed memory programs, dependences result in communication among different processes

Automatic parallelization

- In the presence of complex access patterns (arrays with non-affine subscripts, pointers, etc.) dependence analysis is often too conservative.
- Dependences on two accesses may not exist every time the accesses are encountered. Some parallelization may e possible but hard to impossible to express statically in code -- must be exploited at runtime.

Dusty deck parallelization impossible for many programs

- But works well for some programs and for vectorization
- The key is how to allow programmers to express parallelism with less effort than fully expressing it
- OpenMP and MPI are two attempts to allow parallelism to be expressed
- Can something better be done?
- We will look at some systems that have tried to do something better.