Loop Parallelization Techniques and dependence analysis

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

When can we run code in parallel?

- Two regions of code can be run in parallel when no dependences exist across statements to be run in parallel

```
a = b + c
x =  y + z
u = a + x
```

```
for  (i = 0; i < n; i++) {
    c[i] = a[i] * b[i] + c[i]
}
```

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?

Need to determine if, and in what order, two references access the same memory location

Then 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
```

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

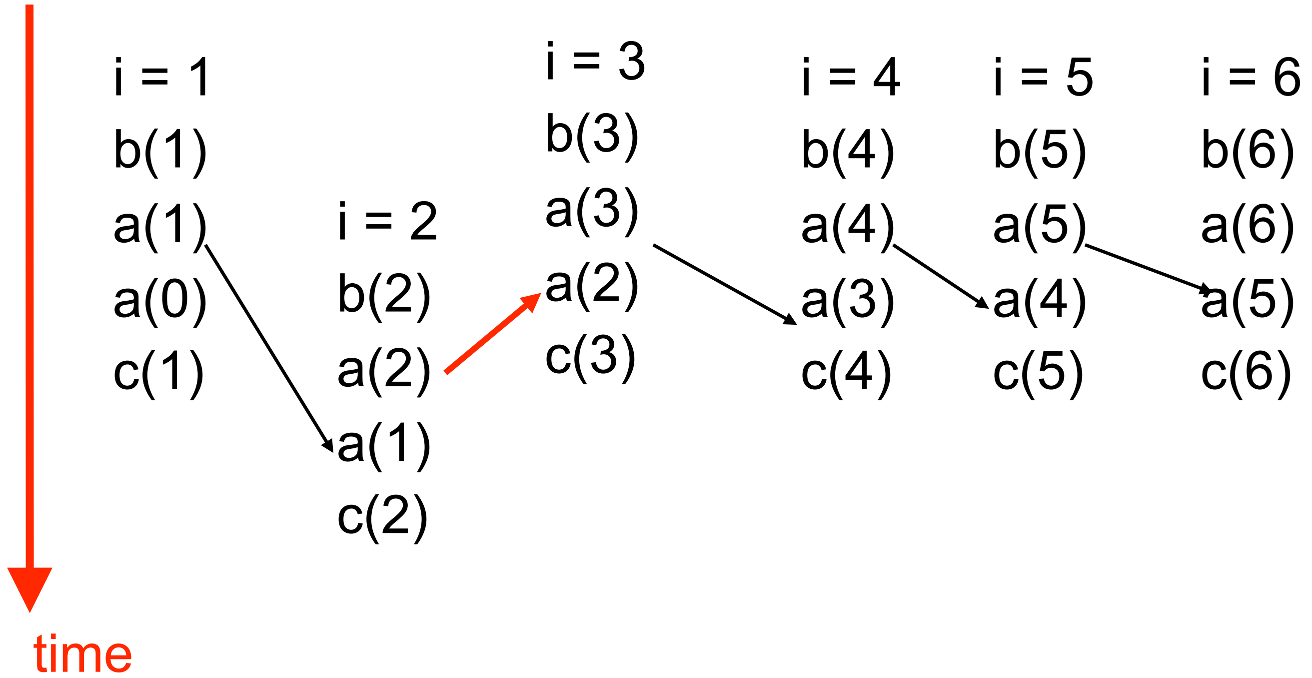
i = 1	i = 2	i = 3	i = 4	i = 5	i = 6
b(1)	b(2)	b(3)	b(4)	b(5)	b(6)
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)
a(0)	a(1)	a(2)	a(3)	a(4)	a(5)
c(1)	c(2)	c(3)	c(4)	c(5)	c(6)

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?

```
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*

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
```

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

In original execution of the unfused loops:

- a(i-1) gets value assigned in a(i)
- Can’t overwrite value assigned to a(i) or c(i)
- B(i) value comes from outside the loop

Types of dependence

a(2) = …
… = a(2)

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

… = a(2)
a(2) = …

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

a(2) = …
a(2) = …

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!

Similar to register renaming

Getting rid of output dependences

a(2) = …

a(2) = …

Output dependence – write to 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.

An example of when it is messy to create new storage

```
do i = 1, n
  a(3i-1) = …
  a(2i) = …
  = … a(i)
end do
```

A(3i) writes locations 2, 5, 8, 11, 14, 17, 20, 23

A(2i) writes locations 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22

A(i) reads from outside the of loop when i = 1, 3, 7, 9, 13, 15, 19, 21

A(i) reads from a(3i-1) when I = 5, 11, 17, 23

A(i) reads from a(2i) when I = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22

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
- Dependences *always go forward in time* in a serial execution

```
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.

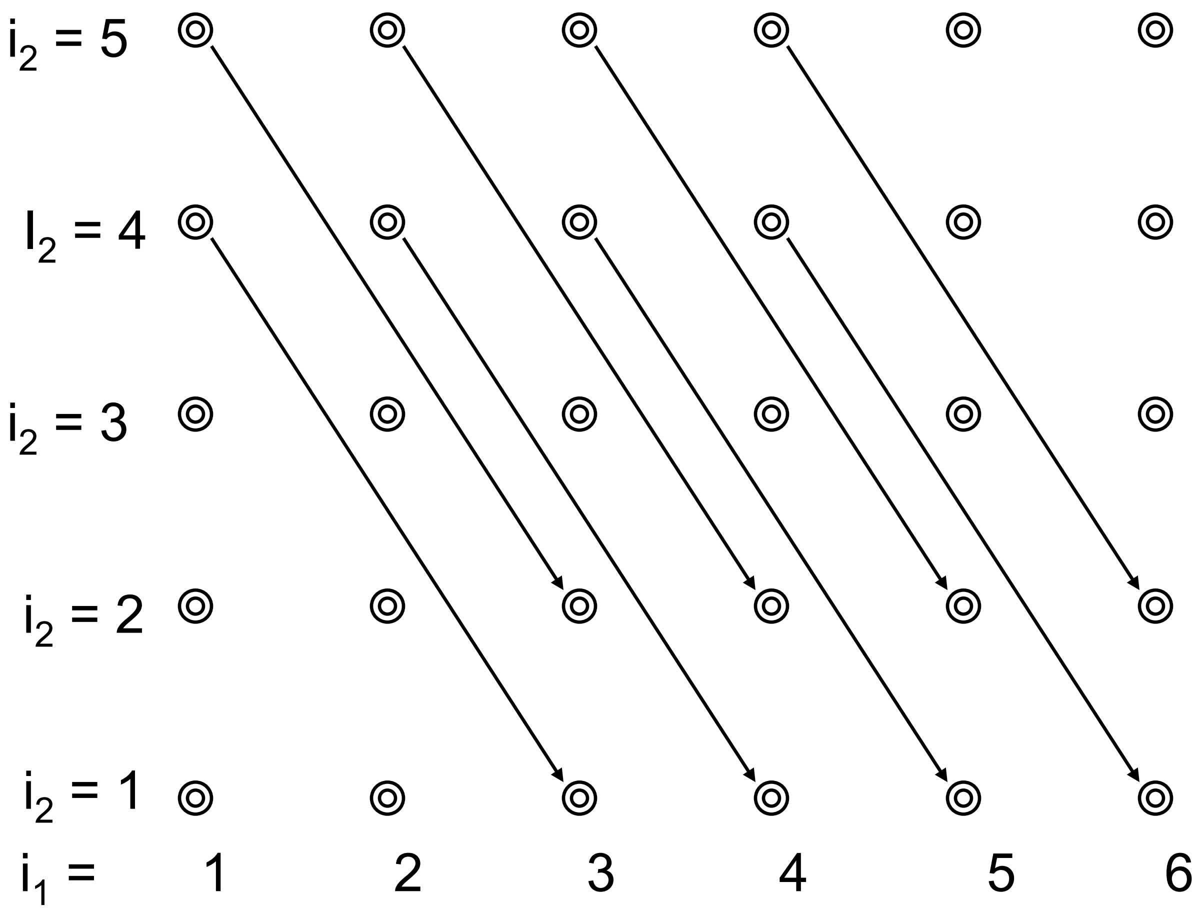
- Distance (vector): indicates how many iterations apart the source and sink of a dependence are.
- Direction (vector): is basically the sign of the distance. There are different notations: (<,=,>) or (+1,0,-1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.
- 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) →



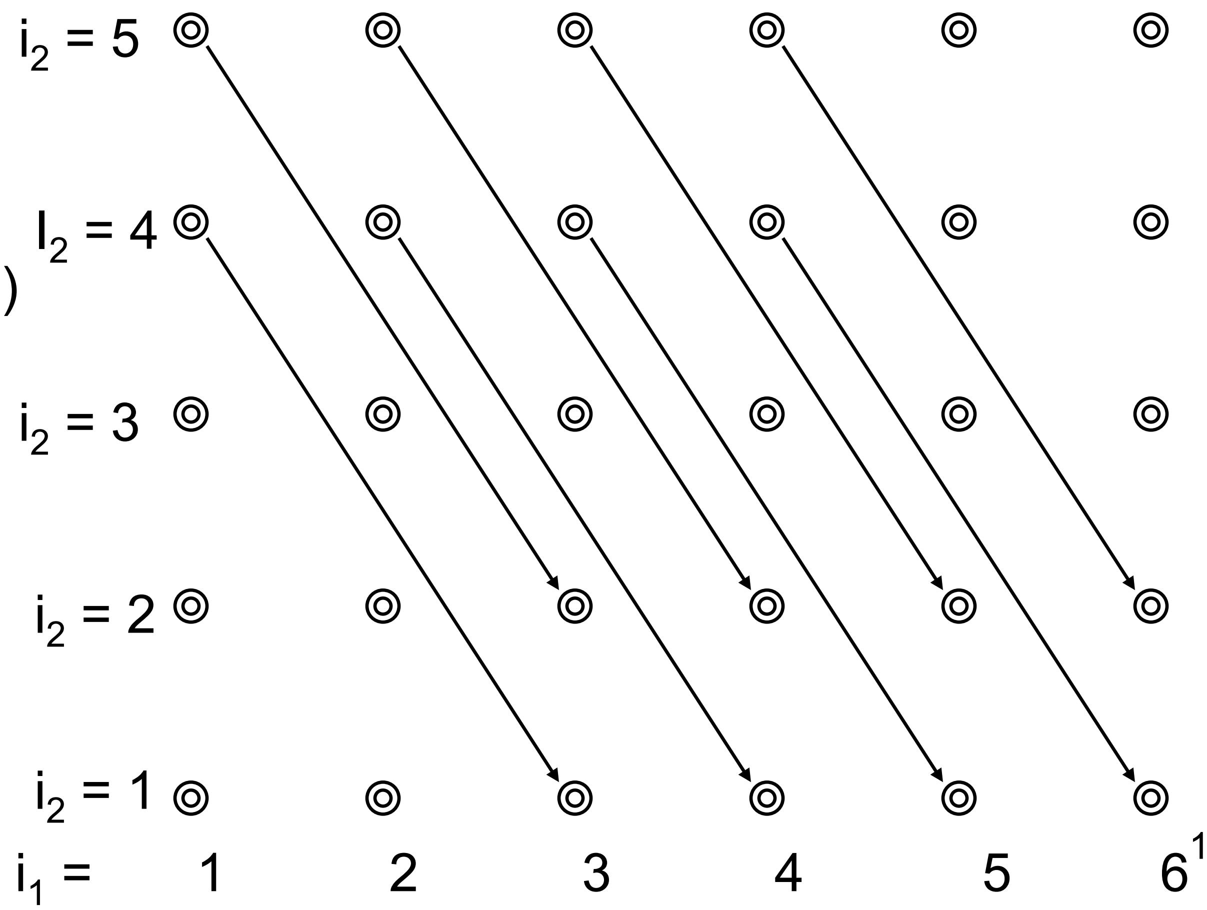
Data Dependence Tests: Distance Vectors

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
```

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

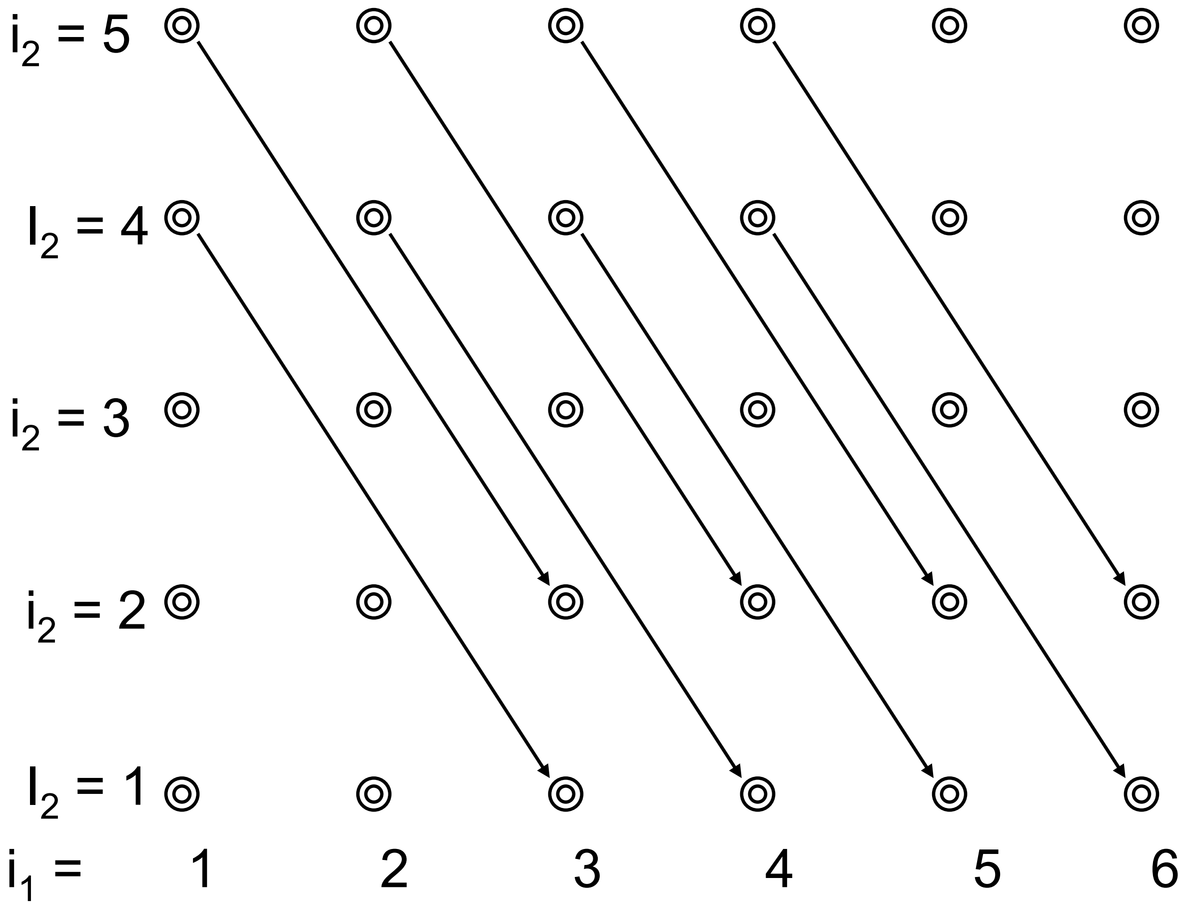
$i' - i = (2,-3)$



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 i_1 = 1, n
  do i_2 = 1, n
    $a(i_1,i_2) = a(i_1-2,i_2+3)$
  end do
end do



I=(1,4) ➔ I' = (3,1)
I' – I = d = (2,-3)
Direction = (<,>) (or sign(2),sign(-3) == (+1,-1)) in some works

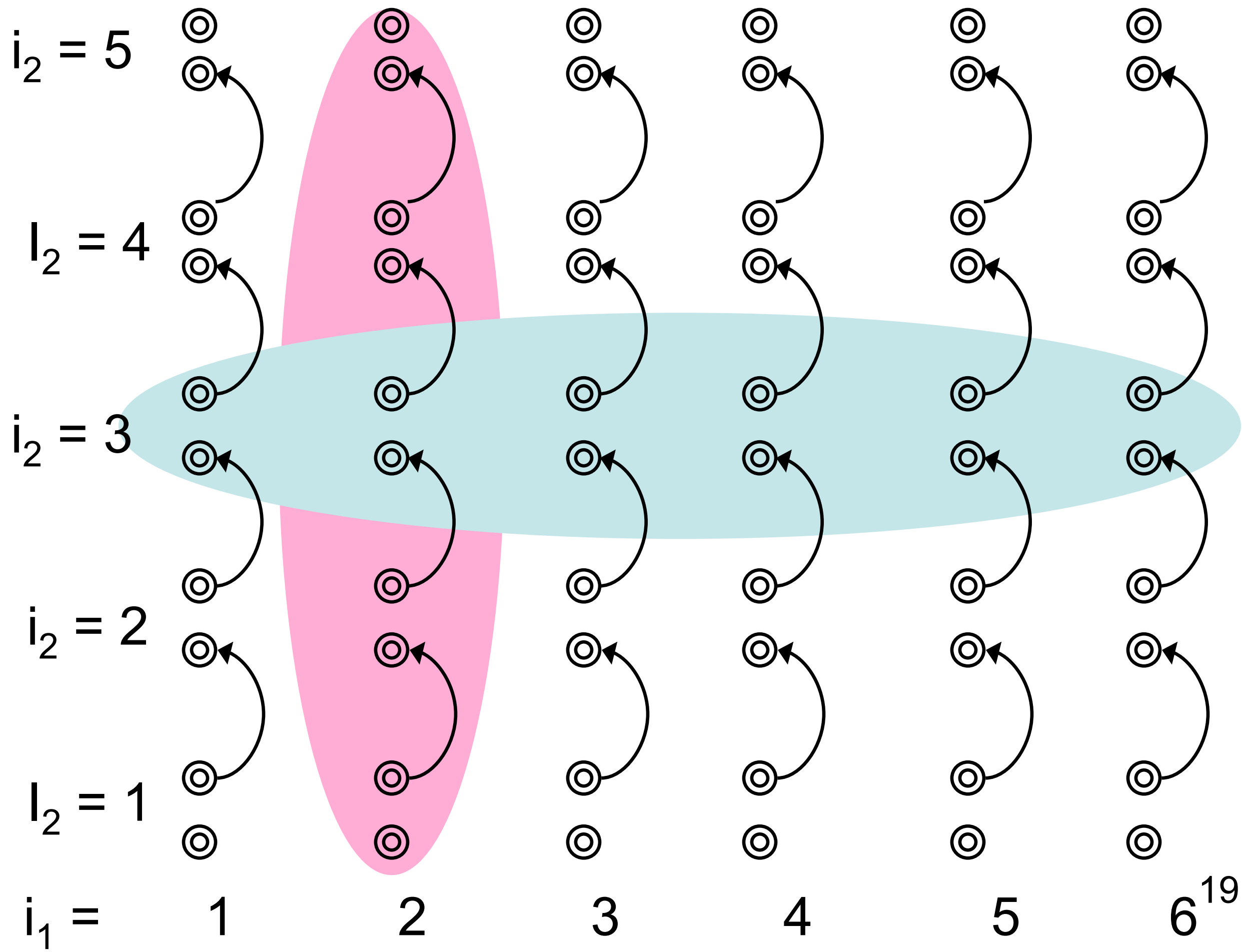
Data Dependence Tests: Loop Carried

- 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.

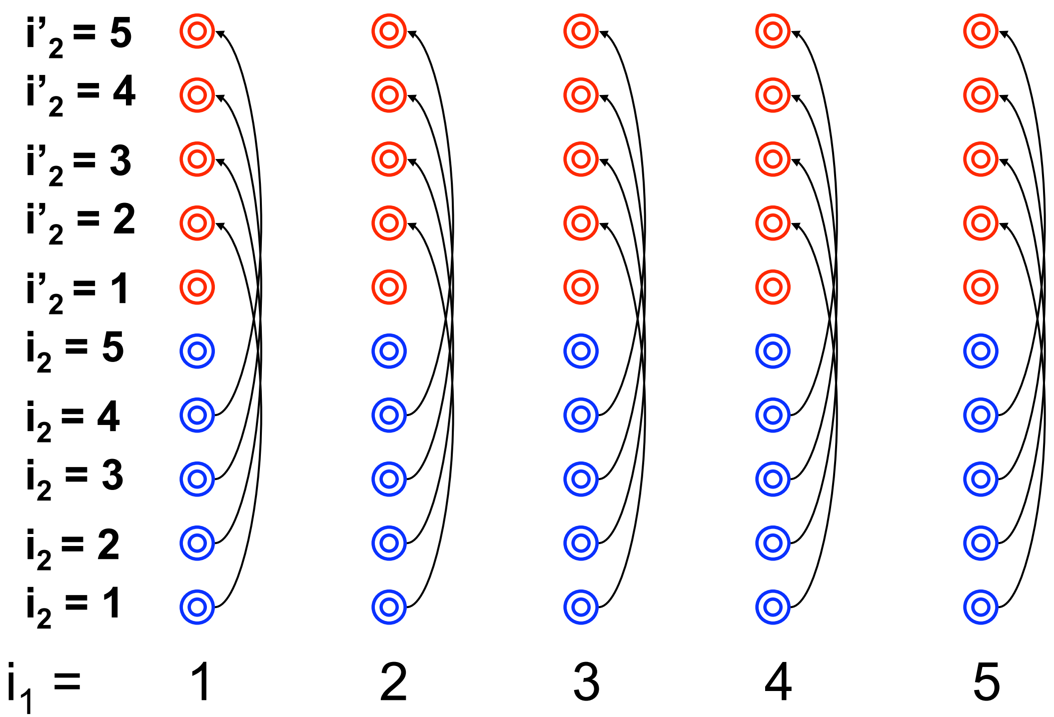


Data Dependence Tests: Loop Carried

- 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
   dopar i2 = 1, n
     a(i1,i2) =
   end
   dopar i'2 = 1, n
           = a(i1,i'2-1)
   end do
end do
```

This is legal since loop splitting enforces the loop carried dependences

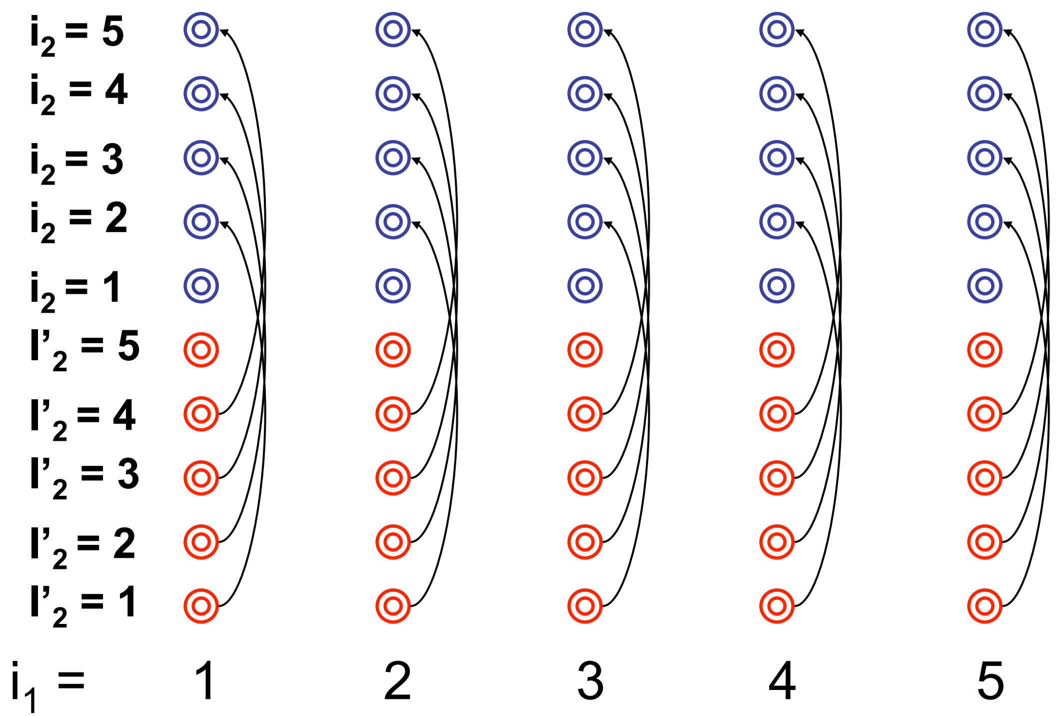


Data Dependence Tests: Loop Carried

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

This is not legal – turns true into anti dependence



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 + stride)/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 within $i \in [1,n]$?
If so, what is the relation of the i's when they are equal?

In general, given:

- two subscript functions *f(I)* and *g(I)* and
- upper and lower loop bounds

Question to answer: Does
f(I) = g(I') have an ***integer*** solution such that
lower $\leq$ *I, I'* $\leq$ *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 $f(i) = a_1*i + c_1$ and $g(i') = b_1*i' + c_2$, then
  - ❖ $f(i) = g(I') \Rightarrow a_1*i - b_1*i' = c_2 - c_1$
  - ❖ fits general form of Diophantine equation of $\boldsymbol{a_1*i_1 + a_2*i_2 = c}$

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

- The Diophantine equation

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

has no solution if $\gcd(a_1, a_2)$ does not evenly divide 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))

Finding GCDs

Euclid Algorithm: find gcd(a,b)
  Repeat
    $a \leftarrow a \bmod b$
    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))

a = 16, b = 6

a ← 16 mod 6

b ← 4, a ← 6

a ← 6 mod 4

b ← 2, a ← 4

a ← 4 mod 2

a ← 2, b ← 0

Determining if a Diophantine equation has a solution

Let $g = gcd(a_1,a_2)$, then 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, (assuming a'_1, a'_2 can range over the integers.)

If remainder(c/g) = 0, c is a solution since $c = g*c'$, and $\cancel{g}*(a'_1*i_1–a'_2*i_2)$ 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 for info on Diophantine equations
- 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 DD Tests

- The GCD test is simple but by itself not very useful
  - Most subscript coefficients are 1, gcd(1,i) = 1
- Other tests
  - ***Banerjee 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 $[lb,ub]$)

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)***

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).

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' (i.e. "=" direction vector), 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.

Cannot parallelize the loop, but can reorder x(i) and x(i-1) within the loop.

Program Transformations

- Applying data dependence tests to untransformed loops would determine that most loops are not parallel.

Reason #1: there are many *anti* and *output dependences*

anti dependence (cross-iter.)

```
DO i=1,n
   t = a(i)+b(I)
   c(i) = t
ENDDO
```

flow dependence (loop-indep.)

Dependence Classification:
- flow dependence:
  read-write dependence
- anti dependence:
  write-read dependence
- output dependence:
  write-write dependence

Solution: *scalar and array privatization*

Scalar Expansion/Privatization

```
DO i = 1,n
  t = a(i) + b(i)
  c(i) = t
ENDDO
```

privatization

```
PARALLEL DO i = 1,n
  Private t
  t = a(i) + b(i)
  c(i) = t
ENDDO
```

expansion

```
PARALLEL DO i = 1,n
  t1(i) = a(i) + b(i)
  c(i) = t1(i)
ENDDO
```

Private creates one copy per parallel loop iteration.

Analysis and Transformation for Scalar Expansion/Privatization

Loop Analysis:

- find variables that are used in the loop body but dead on entry. i.e., the variables are written (on all paths) through the loop before being used.
- determine if the variables are live out of the loop (make sure the variable is defined in the last loop iteration).

Transformation (variable *t*)

- Privatization:
  - put *t* on private list. Mark as *last-value* if necessary.
- Expansion:
  - declare an array *t0(n),* with *n=#loop_iterations.*
  - replace all occurrences of *t* in the loop body with *t0(i),* where *i* is the loop variable.
  - live-out variables: create the assignment *t=t0(n)* after the loop.

Parallelization of Reduction Operations

```
DO i = 1,n
  sum = sum + a(i)
ENDDO
```

```
PARALLEL DO i = 1,n
  ATOMIC:
  sum = sum + a(i)
ENDDO
```

```
PARALLEL DO i = 1,n
  Private s = 0
  s = s + a(i)
POSTAMBLE
  Lock
    sum = sum + s
  Unlock
ENDDO
```

```
DIMENSION s(#processors)
DO j = 1,#processors
  s(j) = 0
ENDDO
PARALLEL DO i = 1,n/#processors
  s(myproc) = s(myproc) + a(i)
ENDDO
DO j = 1,#processors
  sum = sum + s(j)
ENDDO
```

Analysis and Transformation for (Sum) Reduction Operations

- Loop Analysis:
  - find reduction statements of the form *s* = *s* + *expr* where expr does not use *s*.
  - discard *s* as a reduction variable if it is used in non-reduction statements.
- Transformation:
  (as shown on previous slide)
  - create private or expanded variable and replace all occurrences of reduction variable in loop body.
  - update original variable with sum over all partial sums, using a *critical section* in a loop *postamble* or a summation after the loop, respectively.

Induction Variable Substitution

```
ind = ind0
DO j = 1,n
   a(ind) = b(j)
   ind = ind+k
ENDDO
```

→

```
ind = ind0
PARALLEL DO j = 1,n
     a(ind0+k*(j-1)) = b(j)
ENDDO
```

Example: string concat

```
j = eosA
do i = 1, b.length
   a(j) = b(i)
   j = j + 1;
end
```

Gives k*j – **k + indo**

This is of the form

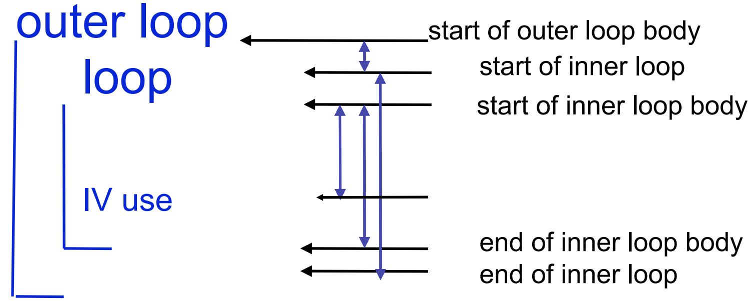
a*j + c, which is good

for dependence analysis. j is the *loop canonical induction variable*.

Induction Variable Analysis and Transformation

- Loop Analyis:
  - find induction statements of the form *s = s + expr* where *expr* is a loop-invariant term or another induction variable.
  - discard variables that are modified in non-induction statements.

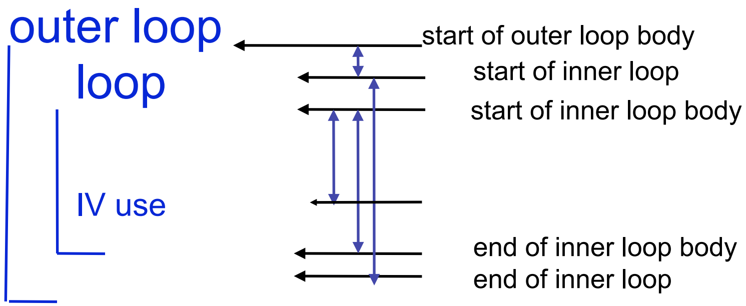
- Transformation:
  - find the following *increments*



  - for each use of IV:
    - compute the increment *inc* with respect to the start of the outermost loop in which it is an induction sequence
    - Replace IV by *inc+ind0*

Induction Variable Analysis and Transformation

- Transformation:
  - find the following *increments*



  - for each use of IV:
    - compute the increment *inc* with respect to the start of the outermost loop in which it is an induction sequence in
    - Replace IV by *inc+ind0*.

```
ind = ind0
PARALLEL DO j = 1,n
    a(ind0+k*(j-1)) = b(j)
ENDDO
```

Thus in the above
- ind0 is obvious;
- inc is k*(j-1)
  - inc is an induction sequence within the loop *DO j*
  - The inner loop body is empty

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

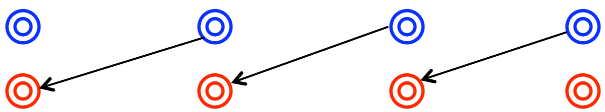
```
DO j = 1,n
  S1: a(j-1) = b(j)
ENDDO

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

fusion →

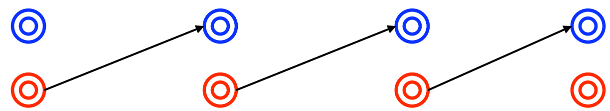
← distribution

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



j, k= 1 2 3 4

Flow dependence from S1 to S2



k = 1 2 3 4

Anti-dependence from S2 to S2

Dependence graphs

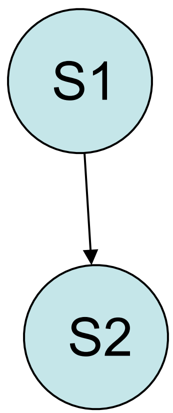
```
DO j = 1,n
  S1: a(j-1) = b(j)
ENDDO

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

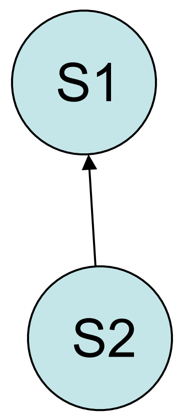
fusion →

← distribution

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



δ^f



$\delta^a(<)$

Loop Interchange

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

→

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

Loop Interchange

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

→

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

- loop interchanging alters the data reference order
  - ➢ significantly affects locality-of reference
  - ➢ data dependences determine the legality of the transformation: dependence structure should stay the same

- loop interchanging may also impact the granularity of the parallel computation (inner loop may become parallel instead of outer)

Loop interchange legality

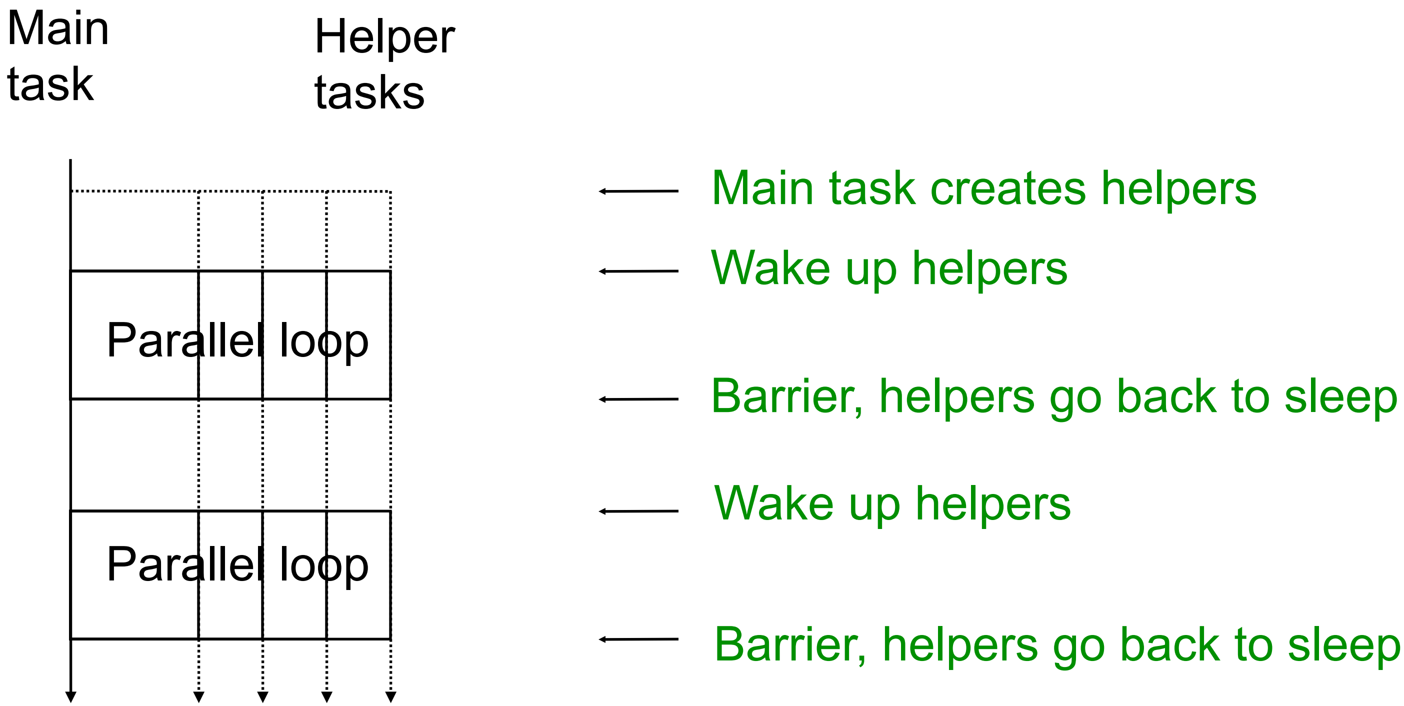
(=,=): after interchange still loop indepent dependences
(=,<): after interchange, is (<,=), still carried on the j loop
(<,=): after interchange is (=,<), still carried on the i loop
(<.<): after interchange still positive in both directions

(>,*), (=,>): not possible – dependences must move forward in Iteration space

(<,>): after interchange is (>,<), except cannot have a (>,<) dependence. The source and sink of the dependence change, changing the dependence. Not legal.

Parallel ExecutionScheme

- Most widely used: Microtasking scheme



Program Translation for

```
   Subroutine x
   ...
C$OMP PARALLEL DO
   DO j=1,n
     a(j)=b(j)
   ENDDO
   …
   END
```

```
Subroutine x
…
call scheduler(1,n,a,b,loopsub)
…
END
```

```
Subroutine loopsub(lb,ub,a,b)
integer lb,ub
DO jj=lb,ub
   a(jj)=b(jj)
ENDDO
END
```