Parallelism Detection

Compilers For High Performance Computing(CHPC)
Master in Research and Innovation (MIRI)

Marc Gonzàlez Tallada
Dept. d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

Index

- **Motivation**
- **Array Section Analysis**
- **Data dependence tests**

Motivation

- In science/engineering applications, loop parallelism is most important source of parallelism.
- If a loop does not have data dependences between any two iterations then it can be safely executed in parallel
- Very simple examples

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

Statement Reordering:
Can these two statements be swapped?

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

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

- A data dependence exists between two data references iff:
  - both references access the same storage location
  - at least one of them is a write access

Motivation

Simple cases

- 1-dimensional array enclosed by a single loop
  - ✓ Can 4*i ever be equal to 2*I+1 within i ∈[1,n] ?

```
DO i=1,1000
  a(4*i) = ...
  ... = a(2*i+1)
END DO
```

$4*i - 2*i = -1$

- 1-dimensional array enclosed by two loops

```
DO i = 1, 100
  DO j = 1, 100
    X(a1*i + b1*j + c1) = ...
    ... = X(a2*i + b2*j + c2)
  END DO
END DO
```

$a_1*i - a_2*i + b_1*j - b_2*j = c_2 - c_1$
$1 \leq i \leq n$
$1 \leq j \leq m$

- In general:
  - ✓ Given two subscript functions *f* and *g* and loop bounds lower, upper.
    - Does *f(i1) = g(i2)* have a solution such that *lower ≤ i1, i2 ≤ upper*

Motivation

- **Simple cases** (cont)
  - 2-dimensional array enclosed by a two loops

```
DO i=1,n
  DO j=1,m
    X(a1*i + b1*j + c1, d1*i + e1*j + f1) = ...
    ... = X(a2*i + b2*j + c2, d2*i +e2*j + f2)
  ENDDO
ENDDO
```

$$a_1*i - a_2*i + b_1*j - b_2*j = c_2 - c_1$$
$$d_1*i - d_2*i + e_1*j - e_2*j = f_2 - f_1$$
$$1 \le i \le n$$
$$1 \le j \le m$$

- **In a general situation with *n* nested loops working with *n-dimensional* data structures**
  - Is there a dependence between S1 and S2 ?

```
   DO i1 = 1, 10
     DO i2 = 1, 100
       ...
       DO in = 1, 100
S1       A( f1(i1, i2, ..., in), f2(i1, i2, ..., in), ..., fn(i1, i2, ..., in) ) = ...
S2       ... = A( g1(i1, i2, ..., in), g2(i1, i2, ..., in), ..., gn(i1, i2, ..., in) )
       END DO
       ...
     END DO
   END DO
```

Motivation

- **Is there a dependence between S1 and S2 ?**
  - A and B might or might not point to the same memory region

```
SUBROUTINE P(A,B,...)
DIMENSION A(NX,NY,NZ), B(NX,NY,NZ)
...
DO i = 1, NX
  DO j = 1, NY
    DO k = 1, NZ
S1    A( f0(i,j,k), f1(i,j,k), f2(i,j,k) ) = ...
S2    ... = B( g0(i,j,k), g1(i,j,k), g2(i,j,k) )
    END DO
  END DO
END DO
END
```

Motivation

- **Is the *k-loop* parallel ?**
  - Subroutine call hides both references
    - ✓ *B(i), A(i,1)*

```
SUBROUTINE P(A,B,...)
DIMENSION A(NX,NY,NZ), B(NX,NY,NZ)
...
DO k = 1, NZ
  CALL COMPUTE(A(1,1,K),B(1,1,K),...,NZ)
END DO
END
```

```
SUBROUTINE COMPUTE(A,B,...,NZ)
DIMENSION A(NX,NY), B(NX)
...
DO i = 1, NZ
  ... = B(i)
  A(i,1) = ...
END DO
END
```

Motivation

- **This course would be pointless if ...**
  - the mathematical formulation of the data dependence problem had an accurate and fast solution, and
  - there were enough loops in programs without any data dependences, and
  - dependence-free code could be executed directly and efficiently.

Array Subscripting Patterns

“Efficient and Precise Array Access Analysis”, Y. Paek, J. Hoeflinger, D Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1
(January 2002) Pages: 65 - 109

Classification of array subscripting patterns

Subscript	Description	Subscript	Description
SS	Subscripted subscripts	CS	Coupled subscripts
NA	Non-affine subscripts	MI	Multi-index subscripts
TA	Affine subscripts in triangular loops	SA	Simple affine subscripts

```
SUBROUTINE sub(n, b, c, z)
INTEGER z[0 : n]
REAL a[0 : 31], b[0:n,0:n], c[0:n]
DO i1 = 0, n, 1
   b(i1, i1) = c(z(i1))
   DO i2 = 0, n, 1
     b(n-i1, i2) = a(8*i1+i2) + a(i1+8*i2+1)
   ENDDO
   DO j2 = 0, i1, 1
     a(i1*(i1+1)/2+j2) = c(j2)
   ENDDO
END DO
END
```

Coupled Subscript (CS): b(i1, i1)

Subscripted subscripts (SS): c(z(i1))

Simple Affine subscript (SA): b(n-i1, i2)

Multi-index subscript (MI): a(8*i1+i2)

Non-Affine subscript (NA): a(i1*(i1+1)/2+j2)

Affine subscript in triangular loop (TA): c(j2)

Array Subscripting Patterns

Classification of array subscripting patterns

subscripts	adm	arc2d	bdna	dyfesm	flo52	mdg	ocean	qcd	swim	tfft2	tomcatv	trfd	turb3d
SS	0.2	2.0	2.9	12.8	0.0	0.0	0.0	1.4	0.0	0.0	0.0	0.0	0.0
NA	0.0	0.0	0.0	0.5	2.8	0.1	0.3	0.0	0.0	3.8	0.0	2.2	0.0
TA	0.0	0.0	0.0	2.1	0.0	0.3	10.6	0.0	0.0	0.0	0.0	0.0	0.0
CS	0.0	0.0	0.0	0.7	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MI	44.4	11.7	0.1	5.2	0.2	3.6	21.4	0.5	0.0	45.8	0.0	14	31.7
SA	55.4	86.3	97.0	78.5	97.0	95.9	67.7	98.1	100	50.4	100	83.8	68.3

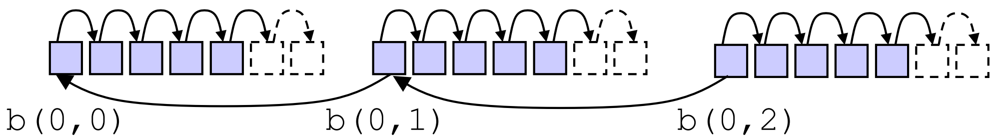
Subscript	Description	Subscript	Description
SS	Subscripted subscripts	CS	Coupled subscripts
NA	Non-affine subscripts	MI	Multi-index subscripts
TA	Affine subscripts in triangular loops	SA	Simple affine subscripts

“Efficient and Precise Array Access Analysis”, Y. Paek, J. Hoeflinger, D Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1
(January 2002) Pages: 65 - 109

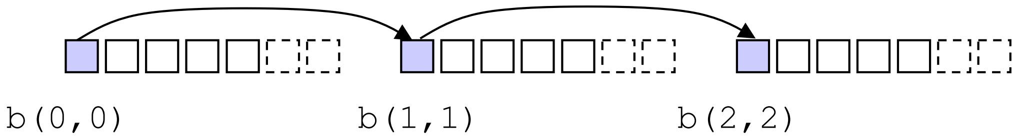
Monotonic Array Accesses

“Efficient and Precise Array Access Analysis”, Y. Paek, J. Hoeflinger, D Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1
(January 2002) Pages: 65 - 109

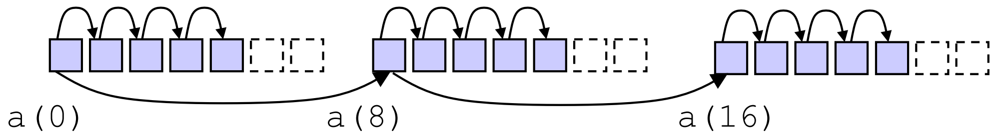
- **b(n-i1, i2)**



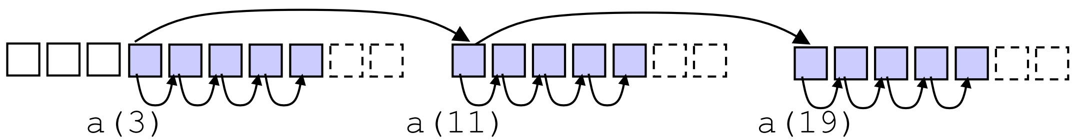
- **b(i1, i1)**



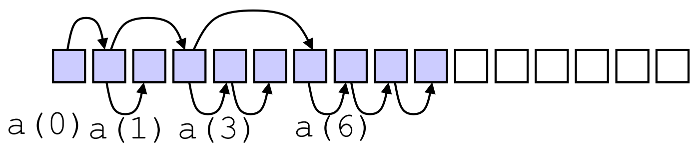
- **a(8*i1+i2)**



- **a(i1+8*i2+1)**



- **a(i1*(i1+1)/2+j2)**

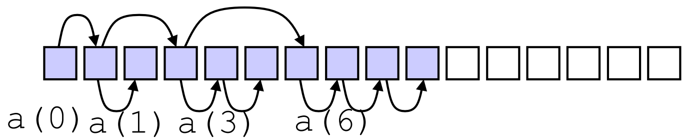


Common Array Accesses

"Efficient and Precise Array Access Analysis", Y. Paek, J. Hoeflinger, D Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1
(January 2002) Pages: 65 - 109

Coalescible Accesses

- Example: *a(i1*(i1+1)/2+j2)*
  - ✓ Makes an access to i_1+1 consecutive elements of array *a* with stride 1 for every iteration of loop index j_2
  - ✓ Then jumps over i_1+1 elements, for every iteration in loop index i_1



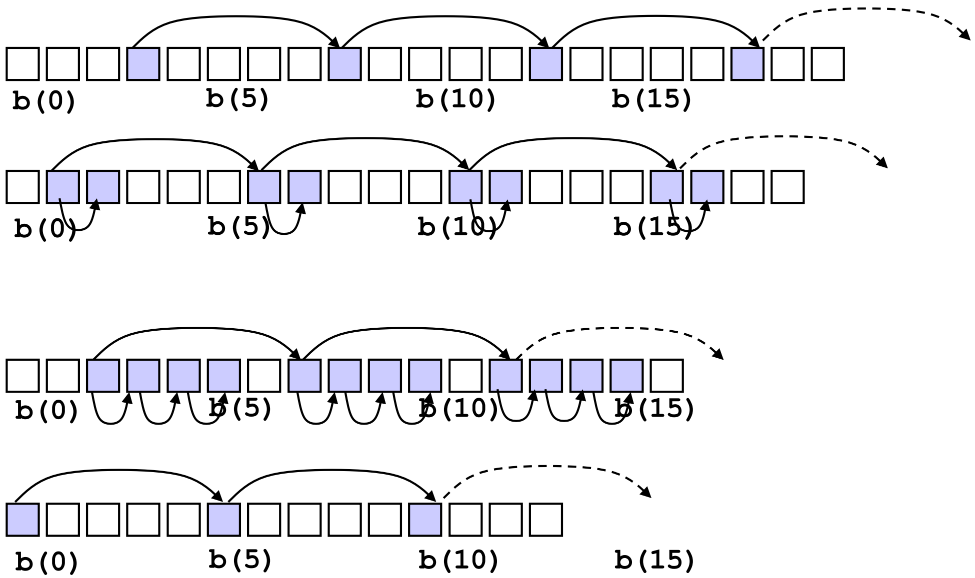
- If an access is coalescible, then the access region generated by multiple indices can be described in terms of one iteration of a single index

Common Array Accesses

“Efficient and Precise Array Access Analysis”, Y. Paek, J. Hoeflinger, D Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1
(January 2002) Pages: 65 - 109

Contiguous Accesses

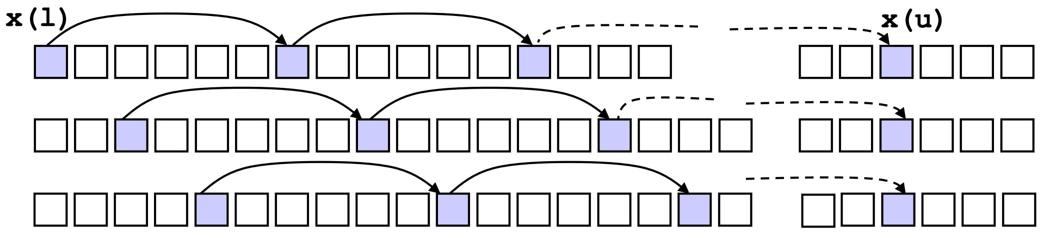
```
REAL b(0:n)
...
DO i = 1, 4, 1
   ... = b(5*i-2)
   DO k = 1,2,1
      ...= b(k+5*(i-1))
   END DO
   DO j = 0, 3, 1
      ... = b(5*i+j-3) - b(5*j)
   END DO
END DO
```



Common Array Accesses

Interleaved Accesses

```
REAL x(0:n), y(0:m)
...
DO i = L, U, 6
   temp = temp + x(i)*y(i)
                + x(i+2)*y(i+2)
                + x(i+4)*y(i+4)
END DO
```



“Efficient and Precise Array Access Analysis”, Y. Paek, J. Hoeflinger, D Padua
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1
(January 2002) Pages: 65 - 109

Array Section Analysis

- **Array sections describe the set of array elements that are either read or written by a program statement**

- **Various factor influence the choice of a particular method**
  - precision
  - efficiency
  - practicality

- **Techniques can be broadly classified into two categories**
  - accurate
  - approximate

Array Section Analysis

- **Reference Lists (accurate)**
- **Linear Constraints (approximate)**
- **Triplet Notation (approximate)**

Reference Lists

- **Accurate method**
- **One descriptor per array reference is constructed**
- **Two implementations**
  - Linearization
  - Atom Images

Linearization

- **Considers the case subscripts that are linear functions of iteration variables**

- **Linear view of the memory**
  - Memory can be viewed as a one dimensional array MEM
  - The function that maps array multiple-subscripts into their locations in MEM is linear with respect to the subscripts
    - ✓ K = LOC(A) – LOC(B)

```
SUBROUTINE P(A, B)
...
DO I = 1, N
  A(I) = ...
  ... = B(I)
END DO
...
END
```

```
SUBROUTINE P(A, B)
...
DO I = 1, N
  MEM(I+K) = ...
  ... = MEM(I)
END DO
...
END
```

"Interprocedural Dependence Analysis and Parallelization", M. Burke, R. Citron
SIGPLAN Symposium on Compiler Construction, July 1986.

Atom Images

"Efficient Interprocedural Analysis for Program parallelization and restructuring", Z. Li and P. Yew
SIGPLAN, June 1988.

- **Considers the case subscripts that are linear functions of iteration variables**
- **Atom definition**
  - Per each array reference one atom is defined
  - Composed by
    - ✓ Name of the array variable
    - ✓ Boolean value indicating if each dimension is addressed through a linear function
    - ✓ Functional description of the function

	linear	const	I_1	I_2	...	I_n
DIM_1	T/F	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	...	$a_{1,n}$
DIM_2	T/F	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	...	$a_{2,n}$
...	T/F	...	...	...	...	...
DIM_n	T/F	$a_{n,0}$	$a_{n,1}$	$a_{n,2}$	...	$a_{n,n}$

```
DO I = 1, 100
  DO J = 1, 100
    A(I+2*J, 2*I+1) = ...
  END DO
END DO
```

	linear	const	I	J
DIM_1	T	1	1	2
DIM_2	T	2	2	0

Linear Constraints

- **A set of linear constraints is constructed per each array reference**
- **The linear constraints refer to the subscripting function and loop bounds**
- **Different representations exist**
  - Restrictions on algebraic properties on the constraints and functions
- **First proposed by**

"Direct Parallelization of Call Statements", Rémi Triolet, Francois Irigoin, Paul Feautrier
Proceedings of the 1986 SIGPLAN symposium on Compiler construction
Palo Alto, California, United States
Pages: 176 - 185

Linear Constraints

- **Simple boundary**
  - Given an *n-dimensional* space with coordinate axes $x_1, x_2, ..., x_n$
  - Hyper-plane of the form
    - ✓ $x_i = c$ or $x_i \pm x_j = c$
- **Simple section**
  - Any convex *polytope* with simple boundaries
- **Most commonly occurring access patterns can be described precisely by a simple section**
  - entire array
  - triangular
  - single diagonal
- **Observation**
  - Simple sections are closed under intersection, but are not closed under union

"A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations"
V. Balasundaram, K. Kennedy
Conference on Programming Language Design and Implementation
Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 - 53

Linear Constraints

"A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations"
V. Balasundaram, K. Kennedy
Conference on Programming Language Design and Implementation
Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 - 53

- **Dominant induction variable**
  - Let $A(x_1, x_2, ..., x_n)$ an array reference contained within m loops with induction variables $I_1, I_2, ..., I_m$
  - Consider the loop at nesting depth *k*, with induction variable I_k $(1 \leq k \leq m)$
  - The dominant induction variable of the *i-th* subscript position of the array reference (with subscript expression x_i) with respect to the I_k loop is determined:
    - ✓ If x_i does not contain any I_s , $k \leq s \leq m$ (i.e., an induction variable of a loop at depth *k* or greater), x_i has no dominant induction variable.
    - ✓ If x_i consists of only one I_s, $k \leq s \leq m$ (i.e., any one induction variable of a loop at depth *k* or greater), I_s is its dominant induction variable.
    - ✓ If x_i contains more than one induction variable belonging to a loop at nesting depth *k* or greater, its dominant induction variable is the induction variable of the loop with the greatest nesting.

Linear Constraints

"A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations"
V. Balasundaram, K. Kennedy
Conference on Programming Language Design and Implementation
Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 - 53

Traversal Order

- Let $A(x_1, x_2, ..., x_n)$ be an array reference contained within *m* loops with induction variables $I_1, I_2, ..., I_m$
- Consider the loop at nesting depth *k*, with induction variable I_k $(1 \leq k \leq m)$
- Let $D_1, D_2, . . . , D_p$, be the list of distinct dominant induction variables of the subscript expressions $x_1, x_2, ..., x_n$, with respect to the I_k loop (in descending order)
- Construct the sets $V_1, V_2, . . . , V_p$ where
  - ✓ $V_i = (x_j, 1 \leq j \leq n \mid D_i$ is its dominant induction variable$)$
  - ✓ If D_i has a negative step size, the $x_j \in V_i$ are denoted as $\neg x_j$, to indicate that access along this dimension occurs in the opposite sense.
- The traversal order of the given array reference with respect to the I_k loop is given by $\tau^{(k)} = V_1 \blacktriangleright V_2 \blacktriangleright ... \blacktriangleright V_p$

Linear Constraints

- **Traversal Order**
  - Example
    - ✓ Consider the outermost loop
      - – $x_1 \rightarrow J$
      - – $x_2 \rightarrow K$
      - – $x_3 \rightarrow J$
    - ✓ This gives the sets
      - – $V_1 = \{\neg x_2\}$
      - – $V_2 = \{x_1, x_3\}$
    - ✓ Traversal order
      - – $V_1 \blacktriangleright V_2$
      - – This traversal order indicates that access along dimension x_2 occurs faster than access along dimensions x_1 and x_3, with access along x_2 occuring in the opposite sense (i.e., from higher to lower values).

```
DO I = 1, 100
  DO J = 100, 1, -1
    DO K = 1, 100
      A(I+J, K+2, J) = ...
    END DO
  END DO
END DO
```

“A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations”
V. Balasundaram, K. Kennedy
Conference on Programming Language Design and Implementation
Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 - 53

Linear Constraints

Data Access Descriptor

- Composed mainly by the simple section and a traversal order
  - ✓ Entire array

```
DO I = 1, 100
  DO J = 1, 100
    A(J,I) = ...
  ENDDO
ENDDO
```

$1 \leq x_1 \leq 100$
$1 \leq x_2 \leq 100$

$V_1 = \{x_1\} \blacktriangleright V_2 = \{x_2\}$

  - ✓ Single column

```
DO I = 1, 100
  DO J = 1, 100
    A(J,I) = ...
  ENDDO
ENDDO
```

$1 \leq x_1 \leq 100$

$V_1 = \{x_1\}$

  - ✓ Single diagonal

```
DO I = 1, 100
  DO J = 1, 100
    A(I,I) = ...
  ENDDO
ENDDO
```

$1 \leq x_1, x_2 \leq 100$
$0 \leq x_1 - x_2 \leq 0$

$V_1 = \{x_1, x_2\}$

  - ✓ Triangular section

```
DO I = 1, 100
  DO J = 1, I
    A(J,I) = ...
  ENDDO
ENDDO
```

$1 \leq x_1, x_2 \leq 100$
$2 \leq x_1 + x_2 \leq 100$
$0 \leq x_1 - x_2 \leq 99$

$V_1 = \{x_2\} \blacktriangleright V_2 = \{x_1\}$

“A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations”
V. Balasundaram, K. Kennedy
Conference on Programming Language Design and Implementation
Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 - 53

Linear Constraints

- **Data Access Descriptor**
  - Intersection
    - ✓ The intersection of two Data Access Descriptors is another Data Access Descriptor
  - Union
    - ✓ The union of two Data Access Descriptors is another Data Access Descriptor

"A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations"
V. Balasundaram, K. Kennedy
Conference on Programming Language Design and Implementation
Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 - 53

Triplet Notation

- **Array subscripting functions can be represented with a set of integer values**
  - (u_k, l_k, s_k)
    - ✓ u_k: Upper bound
    - ✓ l_k: Lower bound
    - ✓ s_k: Stride
- **Example**

```
f0(i,j,k) = i ( 1 : NX : 1)
f1(i,j,k) = j ( 1 : NY : 1)
f2(i,j,k) = k ( 1 : NZ : 1)
```

```
SUBROUTINE P(A,B,...)
DIMENSION A(NX,NY,NZ), B(NX,NY,NZ)
...
DO i = 1, NX
  DO j = 1, NY
    DO k = 1, NZ
      A( f0(i,j,k), f1(i,j,k), f2(i,j,k) ) = ...
      ... = B( g0(i,j,k), g1(i,j,k), g2(i,j,k) )
    END DO
  END DO
END DO
END
```

Index

- Motivation
- Array Section Analysis
- Data dependence tests

Background

Concepts

- Consider two statements S_x and S_y and the multiprocessing of some of the loops that surround S_x and S_y
  - ✓ A *flow-dependence* exists from statement S_x to statement S_y
    - – S_x computes and writes data that can subsequently be read by S_y
  - ✓ An *anti-dependence* exists from statement S_x to statement S_y
    - – S_x reads data from a location into which S_y can subsequently write
- More formally
  - ✓ f and g subscript functions
  - ✓ Instance of S_x with $i_k = x_k$
  - ✓ Instance of S_y with $i_k = y_k$
  - ✓ Example of dependence
    - – flow-dependence

      $[(x_1, x_2) = (3,2) \mid (y_1, y_2) = (5, 1)]$

```
DO i1 = 1, N1
  ...
  DO id = 1, Nd
Sx  X( f(i1, ..., id) ) = ...
Sy  ... = X( g(i1, ..., id) )
  ENDDO
  ...
ENDDO
```

Background

- **Concepts**
  - **Terms for data dependences between statements of loop iterations.**
    - ✓ Distance (vector): indicates how many iterations apart are 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.

Background

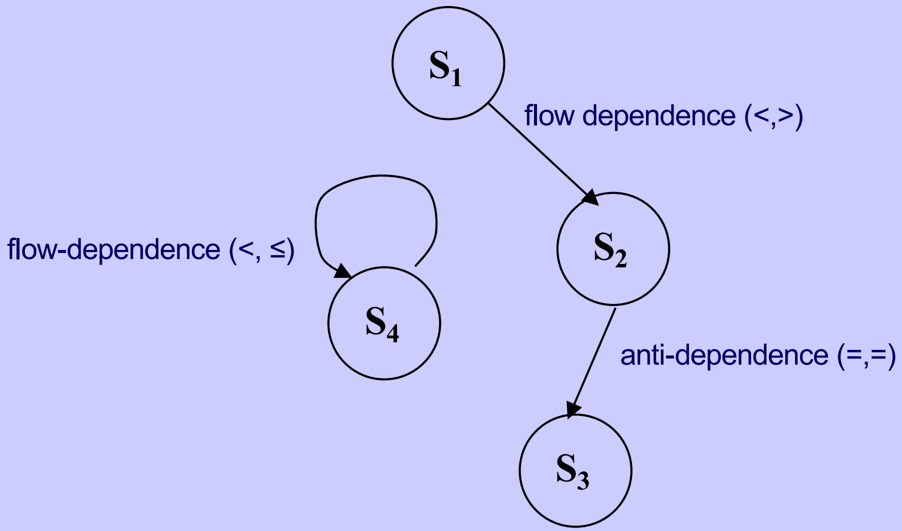
- More formally (cont)
  - Dependence exists
    - ✓ $f(i_1, ..., i_d) - g(i_1, ..., i_d) = 0$
  - f and g linear subscript functions
    - ✓ $f(x_1, ..., x_d) = \Sigma a_k*x_k + a_0$
    - ✓ $g(y_1, ..., y_d) = \Sigma b_k*y_k + b_0$
    - ✓ $f(x_1, ..., x_d) - g(y_1, ..., y_d) = \Sigma(a_k*x_k-y_k*b_k)+a_0- b_0 = 0$
  - Vector direction
    - ✓ $Z = (z_1, ..., z_d)$
      - – z_k = "=", the dependence holds within the same iteration of loop k
      - – z_k = "<", the dependence holds from some iteration of loop k to a subsequent iteration
      - – z_k = ">", the dependence holds from some iteration of loop k to an earlier iteration

```
     DO i1 = 1, N1
       ...
       DO id = 1, Nd
Sx       X( f(i1, ..., id) ) = ...
Sy       ... = X( g(i1, ..., id) )
       ENDDO
       ...
     ENDDO
```

Background

- Example

```
DO i = 1, N
  DO j = 1, M
S1  A(i,j) = ...
S2  ... = A(i-1,j+1)
S3  A(i-1, j+1) = ...
S4  B(i,j) = B(i-1, j-1) + B(i-1,j)
  ENDDO
ENDDO
```



Data Dependence Tests

- **Simple case**
  - **1-dimensional array enclosed by a single loop**
    - ✓ Can 4*i ever be equal to 2*I+1 within i ∈[1,n] ?

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

  - Note: variables i1, i2 are integers → Diophantine equations.
  - Equation $a * i1 - b * i2 = c$ has a solution if and only iff *gcd(a,b) (evenly) divides c*
    - ✓ In our example this means: gcd(4,2)=2, which does not divide 1 and thus there is no dependence
  - If there is a solution, we can test if it lies within the loop bounds. If not, then there is no dependence.

Performing the GCD Test

- **The Diophantine equation a1*i1 + a2*i2 +...+ an*in = c has a solution iff gcd(a1,a2,...,an) 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

- Euklid 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))
```

Other Data Dependence Tests

- **The GCD test is simple but not accurate**
- **Other tests**
  - Banerjee test: accurate state-of-the-art test
  - Omega test: “precise” test, most accurate for linear subscripts
  - Range test: non-linear and symbolic subscripts
    - ✓ many variants of these tests

Banerjee(-Wolfe) Test

- **Basic idea:**
  - if the total subscript range accessed by *ref1* does not overlap with the range accessed by *ref2*, then *ref1* and *ref2* are independent.
  - Example:

```
   DO j=1,100
S1  a(j) = ...
S2  ...= a(j+200)
   ENDDO
```

ranges accesses:
S1 → [1:100]
S2 → [201:300]
Independent!

- **We did not take into consideration that only loop carried dependences matter for parallelization**
  - Example

```
   DO j=1,100
S1  a(j) = ...
S2  ... = a(j+5)
   ENDDO
```

ranges accesses:
S1 →[1:100]
S2 →[6:105]
Independent!

Banerjee(-Wolfe) Test

- **Ranges accessed by iteration j_1 and any other iteration j_2, where $j_1 < j_2$:**
  - S1 → $[j_1]$
  - S2 → $[j_1+6:105]$
  - Independent for “>” direction

```
DO j=1,100
  a(j) = ...
  ... = a(j+5)
ENDDO
```

- **Solution**
  - For loop-carried dependences rely on the fact that j in *ref2* is greater than in *ref1*
  - Clearly, this loop has a dependence. It is an anti-dependence from a(j+5) to a(j)
- **This is commonly referred to as the *Banerjee test with direction vectors.***

Other Data Dependence Tests

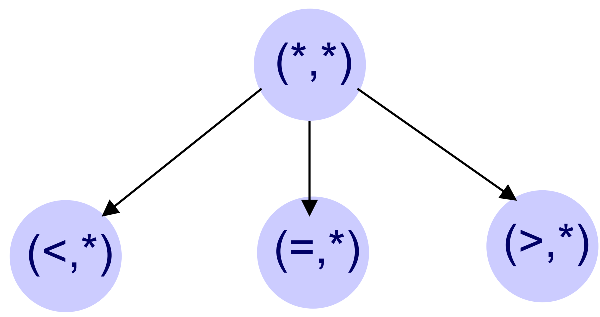
DD Testing with Direction Vectors

- Considering direction vectors can increase the complexity of the DD test substantially.
- For long vectors (corresponding to deeply-nested loops), there are many possible combinations of directions.
- A possible algorithm:
  - ✓ try (*,*…*) , i.e., do not consider directions
  - ✓ (if not independent) try (<,*,*…*), (=,*,*…*)
  - ✓ (if still not independent) try (<,<,*…*),(<,>,*…*) ,(<,=,*…*) (=,<,*…*),(=,>,*…*) ,(=,=,*…*)
  - ✓ …
  - ✓ This forms a tree

The Banerjee(-Wolfe) Test

- Example
  - $(-x_1 + 2*y_1) + (1000*x_2 -100*y_2) = -93$
    - ✓ K = -93
  - (*, *)
    - ✓ -98 < K < 10099
      - – A dependence might exist
  - (<, *)
    - ✓ 3 ≤ K ≤ 10099
      - – No dependence is possible
  - (=, *)
    - ✓ 1 ≤ K ≤ 10000
      - – No dependence is possible
  - (>,*)
    - ✓ -98 ≤ K ≤ 9998
      - – A dependence might exist

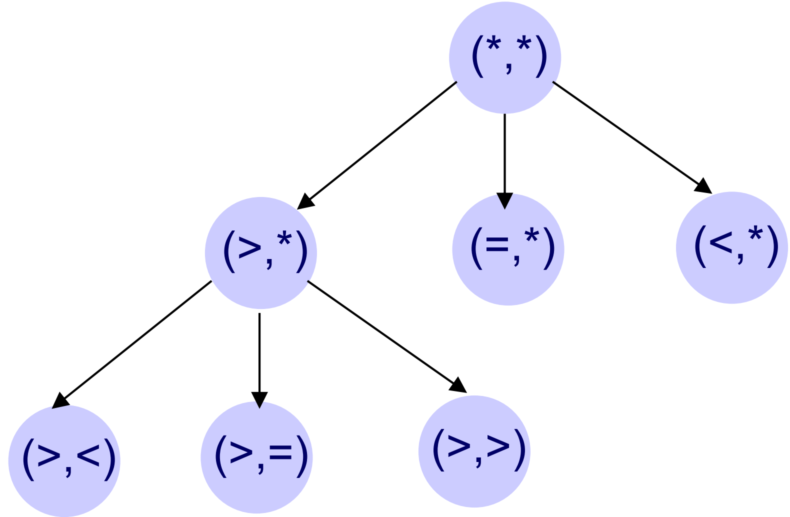
```
   DO i1 = 1, 100
     DO i2 = 1, 10
S1     A(-i1+1000*i2+294) = ...
S2     ... = A(-2*i1-100*i2+201)
     ENDDO
   ENDDO
```



The Banerjee(-Wolfe) Test

```
   DO i1 = 1, 100
     DO i2 = 1, 10
S1     A(-i1+1000*i2+294) = ...
S2     ... = A(-2*i1-100*i2+201)
     ENDDO
   ENDDO
```

- Example
  - $(-x_1 + 2*y_1) + (1000*x_2 - 100*y_2) = -93$
    - ✓ $K = -93$
  - (>,*)
    - ✓ $-98 \le K \le 9998$
      - – A dependence might exist
  - (>,<)
    - ✓ $-98 \le K \le 8098$
      - – A dependence might exist
  - (>,=)
    - ✓ $802 \le K \le 9098$
      - – No dependence is possible
  - (>,>)
    - ✓ $1702 \le K \le 9998$
      - – No dependence is possible



Non-linear and Symbolic DD Testing

- **Weakness of most data dependence tests:**
  - subscripts and loop bounds must be affine, i.e., linear with integer-constant coefficients
- **Approach of the Range Test:**
  - capture subscript ranges symbolically
  - compare ranges: find their upper and lower bounds by determining *monotonicity*.
  - Monotonically increasing/decreasing ranges can be compared by comparing their upper and lower bounds

The Range Test

- **Basic idea :**
  - Find the range of array accesses made in a given loop iteration
  - If the upper (lower) bound of this range is less (greater) than the lower (upper) bound of the range accesses in the next iteration, then there is no cross-iteration dependence.
- **Example:**
  - Testing independence of the outer loop

```
DO i=1,n
  DO j=1,m
    A(i*m+j) = 0
  ENDDO
ENDDO
```

range of A accessed in iteration i_x: $[i_x*m+1:\overbrace{(i_x+1)*m}^{UB_x}]$

range of A accessed in iteration i_x+1: $[\underbrace{(i_x+1)*m+1}_{LB_{x+1}}:(i_x+2)*m]$

$UB_x < LB_{x+1} \Rightarrow$ no cross-iteration dependence

The Range Test

- **Assume *f,g* are monotonically increasing for all i_x:**
  - find upper bound of access range at loop k: successively substitute i_x with U_x, x={n,n-1,...,k} lower bound is computed analogously
- **If *f,g* are monotonically decreasing for all i_y, then substitute L_y when computing the upper bound.**

```
DO i1=L1,U1
  ...
  DO in=Ln,Un
    A(f(i0,...in)) = ...
    ... = A(g(i0,...in))
  ENDDO
  ...
ENDDO
```

The Range Test

- **Determining monotonicity:**
  - Consider $d = f(\ldots,i_k,\ldots) - f(\ldots,i_k-1,\ldots)$
  - If $d>0$ (for all values of i_k) then f is monotonically increasing w.r.t. k
  - If $d<0$ (for all values of i_k) then f is monotonically decreasing w.r.t. k
- **What about symbolic coefficients?**
  - In many cases they cancel out
  - If not, find their range (i.e., all possible values they can assume at this point in the program), and replace them by the upper or lower bound of the range.

References

- **Array Section Analysis**
  - "Efficient and Precise Array Access Analysis", Y. Paek, J. Hoeflinger, D Padua ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 24 , Issue 1 (January 2002) Pages: 65 - 109
  - "Efficient Interprocedural Analysis for Program parallelization and restructuring", Z. Li and P. Yew SIGPLAN, June 1988.
  - "A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations" V. Balasundaram, K. Kennedy Conference on Programming Language Design and Implementation Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation Portland, Oregon, United States Pages: 41 – 53
  - "Interprocedural Dependence Analysis and Parallelization", M. Burke, R. Citron SIGPLAN Symposium on Compiler Construction, July 1986.
  - "Direct Parallelization of Call Statements", Rémi Triolet, Francois Irigoin, Paul Feautrier Proceedings of the 1986 SIGPLAN symposium on Compiler construction Palo Alto, California, United States Pages: 176 - 185

References

- **Data Dependence Test**
  - Banerjee/Wolfe test
    - ✓ M.Wolfe, U.Banerjee, "Data Dependence and its Application to Parallel Processing", Int. J. of Parallel Programming, Vol.16, No.2, pp.137-178, 1987
  - Range test
    - ✓ William Blume and Rudolf Eigenmann. Non-Linear and Symbolic Data Dependence Testing, IEEE Transactions of Parallel and Distributed Systems, Volume 9, Number 12, pages 1180-1194, December 1998.
  - Omega test
    - ✓ William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence. Proceedings of the 1991 ACM/IEEE Conference on Supercomputing,1991
  - I Test
    - ✓ Xiangyun Kong, David Klappholz, and Kleanthis Psarris, "The I Test: A New Test for Subscript Data Dependence," Proceedings of the 1990 International Conference on Parallel Processing, Vol. II, pages 204-211, August 1990.

END