RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS 516 Compilers and Programming Languages II

Parallel Computing-5

Annoucements

Project 1 deadline extension: Wednesday, April 1 (no joke ☺ !)
Can give longer extension if needed

Power Strips: File systems are RAMDisks, which means that when rebooting, all information was lost; script will be reinstalled

Homework #3 has been posted; will cover vectorization algorithms this week

First paper presentation starting second week in April
- what's next
- polyhedral parallelisation

Simple Dependence Testing

```
DO i1 = L1, U1
     DO i2 = L2, U2
        ...
          DO in = Ln, Un
     S1     A(f1(i1,...,in),...,fm(i1,...,in)) = ...
     S2       ... = A(g1(i1,...,in),...,gm(i1,...,in))
          ENDDO
        ...
     ENDDO
ENDDO
```

A dependence exists from S_1 to S_2 if and only if there exist values of α and β such that

(1) α is lexicographically less than or equal to β ($\alpha \leq \beta$), and

(2) the following **system of dependence equations** is satisfied:

$$f_i(\alpha) = g_i(\beta) \text{ for all } i,\ 1 \leq i \leq m$$

Dependence Testing

Can we solve this problem exactly?
What is conservative in this framework? (false positive vs. false negative)

Typically: restrict the problem to consider index and bound expressions that are linear functions

$\Rightarrow$ solving general system of linear equations in integers is NP-hard

Solution Methods

Inexact methods

- Greatest Common Divisor (GCD)
- Banerjee's inequalities

Cascade of exact, efficient tests (fall back on inexact methods if needed)

- Rice (see posted PLDI'91 paper)
- Stanford

Polyhedral dependence representation

Exact general tests (integer programming)

Simple Dependence Testing: Delta Notation

- Notation represents index values at the source and sink

  Example:

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

- Iteration at source denoted by: I_0 (α)
- Iteration at sink denoted by: $I_0 + \Delta I$ (β)
- Forming an equality gets us: $I_0 + 1 = I_0 + \Delta I$
- Solving this gives us: $\Delta I = 1$
  - $\Rightarrow$ Carried dependence with distance vector (1) and direction vector (<)

Simple Dependence Testing: Delta Notation

Example:

```
DO I = 1, 100
    DO J = 1, 100
        DO K = 1, 100
            A(I+1,J,K) = A(I,J, K+1) + B
        ENDDO
    ENDDO
ENDDO
```

- $I_0 + 1 = I_0 + \Delta I$; $J_0 = J_0 + \Delta J$; $K_0 = (K_0 + \Delta K + 1)$
- Solutions: $\Delta I = 1$; $\Delta J = 0$; $\Delta K = -1$
- Corresponding direction vector: (<, =, >)
- Corresponding distance vector: (1, 0, -1)

Dependence Test: SIV (single induction variable)

```
DO I = LB, UB, 1
R1:   A(a*I+c1) = ...
R2:        ... = A(a*I+c2)
ENDDO
```

- constant loop bounds LB and UB, step is 1
- I is single loop induction variable
- a, c_1 and c_2 are constants, $a \neq 0$

There is a dependence between R_1 and R_2 iff

$$\exists i, i' : i \leq i' \;\; and \;\; (a * i + c_1) = (a * i' + c_2)$$

So let's solve the equation:

$$(a * i + c_1) = (a * i' + c_2) \Leftrightarrow$$

$$\frac{c_1 - c_2}{a} = i' - i = \Delta d$$

There is a dependence between R_1 and R_2 with distance Δd iff

(1) Δd is an integer value

(2) UB – LB $\geq \Delta$d $\geq$ 0

Dependence Test: SIV (single induction variable)

Examples:

```
DO I = LB, UB, 1
R1:    X(I) = ...           // write
R2:       ... = X(I-2)      // read
ENDDO
```

```
DO I = LB, UB, 1
R1:    X(2*I) = ...           // write
R2:       ... = X(2*I-1)      // read
ENDDO
```

Dependence Test: SIV (single induction variable)

Examples:

```
DO I = LB, UB, 1
R1:    X(I) = ...          // write
R2:       ... = X(I-2)     // read
ENDDO
```

$a = 1,\ c_1 = 0,\ c_2 = -2 \ \Rightarrow \Delta d = 2$ (dependence)

```
DO I = LB, UB, 1
R1:    X(2*I) = ...          // write
R2:       ... = X(2*I-1)     // read
ENDDO
```

$a = 2,\ c_1 = 0,\ c_2 = -1 \ \Rightarrow \Delta d = \frac{1}{2}$ (no dependence)

Dependence Test: ZIV (zero induction variable)

```
DO I = LB, UB, 1
R1:    A(c1) = ...
R2:       ... = A(c2)
ENDDO
```

- constant loop bounds LB and UB, step is 1
- I is single loop induction variable
- c_1 and c_2 are constants

There is a dependence between R_1 and R_2 iff

$$c_1 = c_2 = c.$$

What about Δd?
Since every iteration i writes *A*(c) and reads *A*(c),

$\Delta d \in \{0, \dots \text{UB-LB}\}$ for true dependence

$\Delta d \in \{1, \dots \text{UB-LB}\}$ for anti and output dependence

Therefore, we summarize as Δd as (*)

Simple Dependence Testing: Delta Notation

If a loop index does not appear, its distance is unconstrained and its direction is "*"

Example:

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

The direction vector for the true dependence is (<, *)

Example: Iteration (3, 2) and iteration (4, 1) both access A(4), with (3,2) writing A(4) and (4, 1) reading A(4): distance (1,-1)
the same holds for iteration (3, 81) and iteration (4, 98): distance (1,17)

Simple Dependence Testing: Delta Notation

Let's swap iterations I and J (loop interchange)

Example:

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

Loop interchange will change the semantics of the loop nest!

Now, we have true, output, and anti dependencies with different distances.

Examples: true – (3, 2) writes A(3), and (10, 3) reads A(3): distance (7, 1)
anti - (3, 2) reads A(2), and (10, 1) writes A(2): distance (7, -1)
output - (3, 2) writes A(3), and (10, 2) writes A(3): distance (7, 0)

Summary: true (< , <) and (=, <); anti (<, <); output (<, <)

Parallelization and Vectorization

Theorem It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

Want to convert loops like:

```
DO I=1,N
    X(I) = X(I) + C
ENDDO
```

to `X(1:N) = X(1:N) + C` (vector notation)

However:

```
DO I=1,N
    X(I+1) = X(I) + C
ENDDO
```

is not equivalent to `X(2:N+1) = X(1:N) + C`

Loop Distribution

Can statements in loops which carry dependences be vectorized?

```
      DO I = 1, N
S1      A(I+1) = B(I) + C
S2      D(I) = A(I) + E
      ENDDO
```

Dependence: $S_1 \, \delta_1 \, S_2$ can be converted to:

```
S1    A(2:N+1) = B(1:N) + C
S2    D(1:N) = A(1:N) + E
```

Loop Distribution

```
    DO I = 1, N
S1       A(I+1) = B(I) + C
S2       D(I) = A(I) + E
      ENDDO
```

transformed to:

```
    DO I = 1, N
S1       A(I+1) = B(I) + C
    ENDDO
    DO I = 1, N
S2       D(I) = A(I) + E
    ENDDO
```

leads to:

```
S1       A(2:N+1) = B(1:N) + C
S2       D(1:N) = A(1:N) + E
```

Loop Distribution

Loop distribution fails if there is a cycle of dependences

```
     DO I = 1, N
S1   A(I+1) = B(I) + C
S2   B(I+1) = A(I) + E
     ENDDO
```

$S_1 \ \delta_1 \ S_2$ and $S_2 \ \delta_1 \ S_1$

What about:

```
     DO I = 1, N
S1      B(I) = A(I) + E
S2      A(I+1) = B(I) + C
     ENDDO
```

Simple Vectorization Algorithm

procedure *vectorize* (L, D)

// L is the maximal loop nest containing the statements.
// D is the dependence graph for statements in L.

find the partition p of set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D restricted to L (for example, using Tarjan's algorithm);

construct L_p from L by reducing each S_i to a single node and compute D_p, the dependence graph naturally induced on L_p by D;

let $\{p_1, p_2, \ldots, p_m\}$ be the m nodes of L_p numbered in an order consistent with D_p (use topological sort);

for i = 1 to m do begin

if p_i is a dependence cycle (excluding loop-carried anti-dependence cycle) then
generate a DO-loop around the statements in p_i;

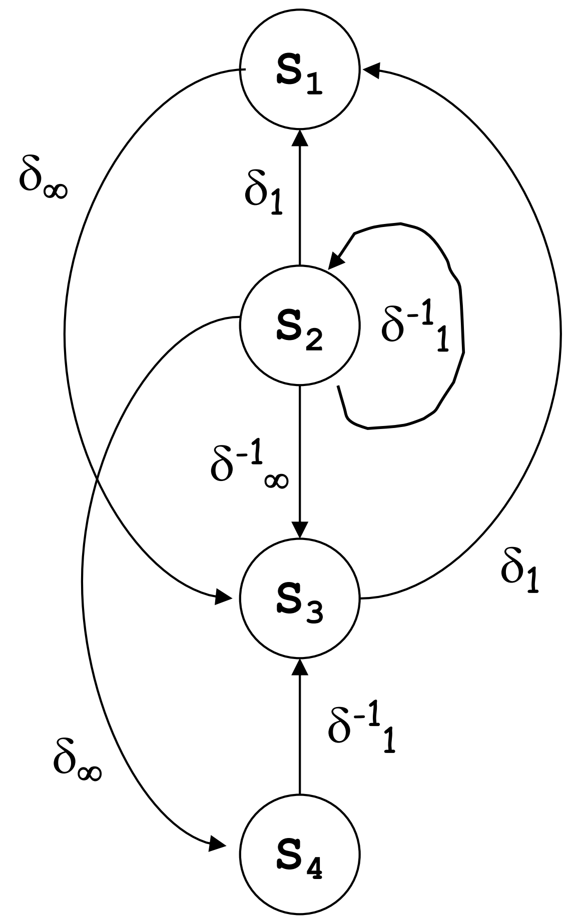
else
directly rewrite p_i in vector notation, vectorizing it with respect to every loop containing it;

end

end *vectorize*

Simple Vectorization Algorithm

Statement level dependence graph D:



Example Loop L:

```
DO I = 2, 100
S1    A(I) = B(I-1) + C(I-1)+ 1
S2    B(I) = C(I)* B(I+1)
S3    C(I) = A(I) + 5
S4    D(I) = B(I)+ C(I+1)
ENDDO
```

δ_k - level k true dependence
δ^o_k - level k output dependence
δ^{-1}_k - level k anti dependence

Loop independent: $k = \infty$

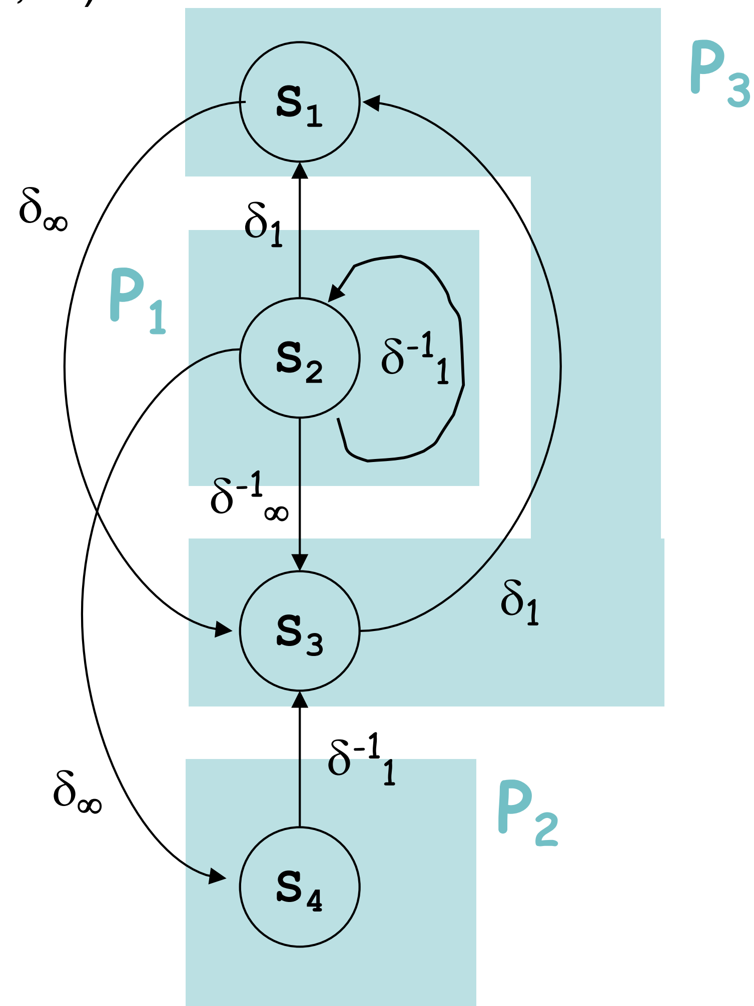
Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

vectorize (*L*, *D*)

```
DO I = 2, 100
S1   A(I) = B(I-1) + C(I-1)+ 1
S2   B(I) = C(I)* B(I+1)
S3   C(I) = A(I) + 5
S4   D(I) = B(I)+ C(I+1)
ENDDO
```



δ_k - level k true dependence

δ^o_k - level k output dependence

δ^{-1}_k - level k anti dependence

Loop independent: $k = \infty$

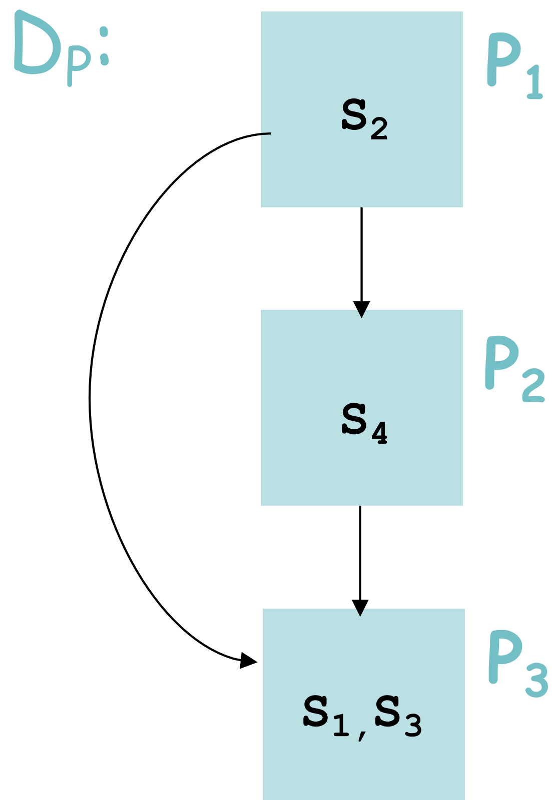
Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

vectorize (*L*, *D*)

```
DO I = 2, 100
S1    A(I) = B(I-1) + C(I-1)+ 1
S2    B(I) = C(I)* B(I+1)
S3    C(I) = A(I) + 5
S4    D(I) = B(I)+ C(I+1)
ENDDO
```



D_P has to be acyclic

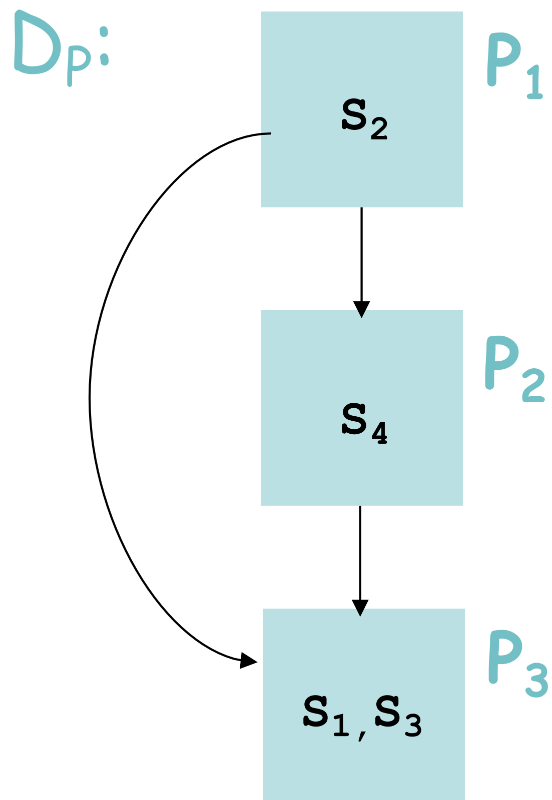
Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

vectorize (*L*, *D*)

```
    DO I = 2, 100
S1     A(I) = B(I-1) + C(I-1)+ 1
S2     B(I) = C(I)* B(I+1)
S3     C(I) = A(I) + 5
S4     D(I) = B(I)+ C(I+1)
    ENDDO
```



In topological order:

```
S2 B(2:100) = C(2:100)* B(3:101)
S4 D(2:200) = B(2:100)+ C(3:101)
   DO I = 2, 100
S1     A(I) = B(I-1) + C(I-1)+ 1
S3.    C(I) = A(I) + 5
   ENDDO
```

D_P has to be acyclic