

并行编译与优化 Parallel Compiler and Optimization

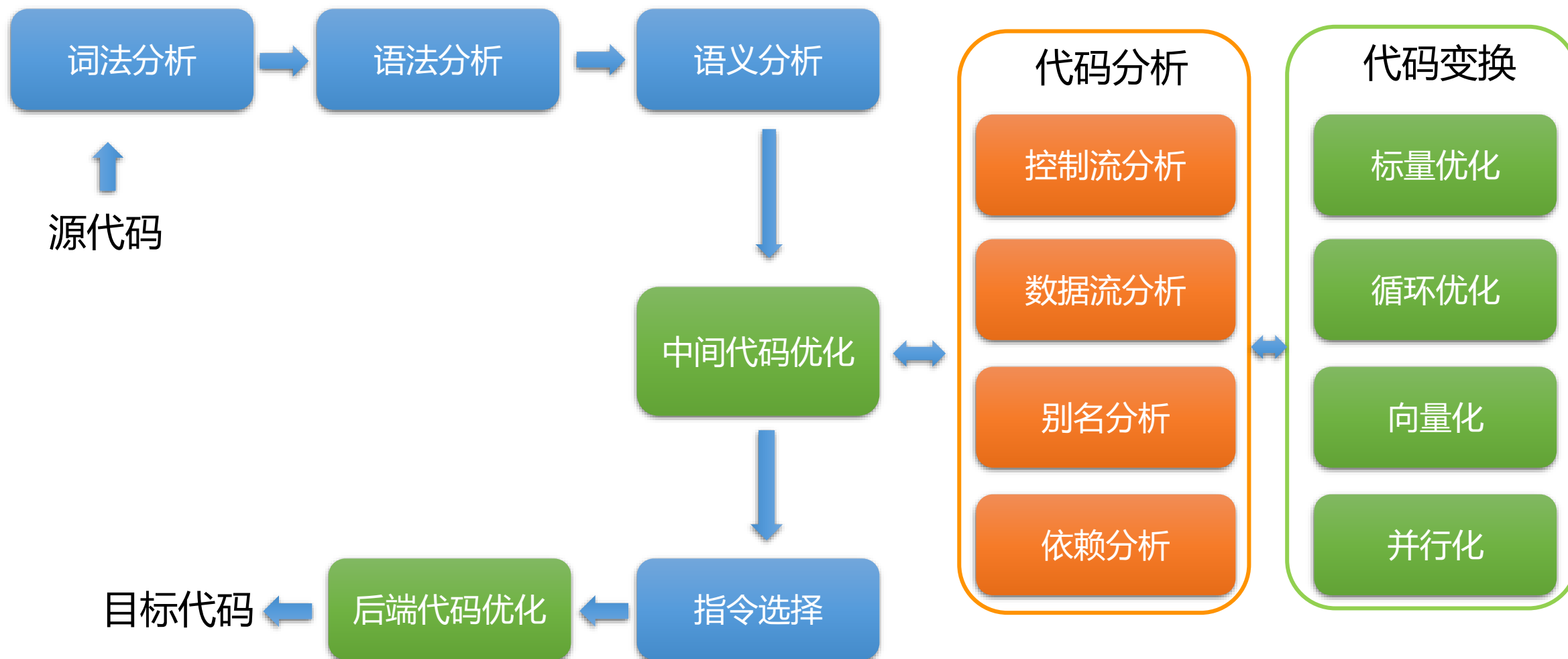
计算机研究所编译系统室 方建滨

Lecture Nine: Dependence Analysis

第九课：依赖关系分析（二）

2025-04-07

回顾：编译流程



■ 1. 依赖关系概念及分类

■ 2. 控制依赖关系

- ⊕ 2.1 控制依赖关系的概念
- ⊕ 2.2 后必经关系
- ⊕ 2.3 控制依赖关系的严格定义
- ⊕ 2.4 控制依赖关系的计算方法
- ⊕ 2.5 课堂小结与作业

■ 3. 数据依赖关系

- ⊕ 3.1 数据依赖关系的概念
- ⊕ 3.2 循环中的数据依赖关系
- ⊕ 3.3 依赖距离、方向、层次
- ⊕ 3.4 数据依赖关系的计算方法
- ⊕ 3.5 课堂小结与作业

■ SPEC CPU是国际公认的评测处理器的基准程序

- ⊕ 浮点计算性能和整数计算性能
- ⊕ **循环代码** 占据了SPEC CPU程序90%以上的执行时间 (**二八定律**)

■ 循环代码是编译优化的主要对象

- ⊕ 并行化: 多线程、多进程
- ⊕ 向量化: 向量单元
- ⊕ 这是超级计算机发挥潜能最主要的手段

■ 数据依赖关系分析是并行化和向量化的基础

- ⊕ **分析** 一个循环是否可被并行化/向量化
- ⊕ 如果不能, 是否有合适的循环 **变换**
- 编译优化

本讲目标: 掌握数据依赖的基本概念和测试方法

■ 深度为m的紧嵌套循环L

```
do i1 = l1, u1, s1
  do i2 = l2, u2, s2
    ∴
    do im = lm, um, sm
      H(i1, i2, ..., im)
    enddo
    ∴
  enddo
enddo
```

i_r : 索引变量

所有索引变量组成索引向量

$I=(i_1, i_2, \dots, i_m)$, 向量长度 m

索引向量 $I < J$, 当且仅当存在一个整数 l , 满足

$i_1=j_1, i_2=j_2, \dots, i_{l-1}=j_{l-1}, i_l < j_l$

■ 深度为m的紧嵌套循环L

```
do i1 = l1, u1, s1
  do i2 = l2, u2, s2
    ∴
    do im = lm, um, sm
      H(i1, i2, ..., im)
    enddo
    ∴
  enddo
enddo
```

循环体H由若干语句组成,

$S(i_1, i_2, \dots, i_m)$ 是语句S一个实例,
令 $I=(i_1, i_2, \dots, i_m)$, 实例简写为S(I)

$T(j_1, j_2, \dots, j_m)$ 是语句T一个实例,
简写为T(J)

■ 数组下标(subscript)

⊕ 一般是循环索引向量的线性表达式

⊕ 以矩阵乘法为例

◆ $C(m, n)$

◆ $A(m, k)$

◆ $B(k, n)$

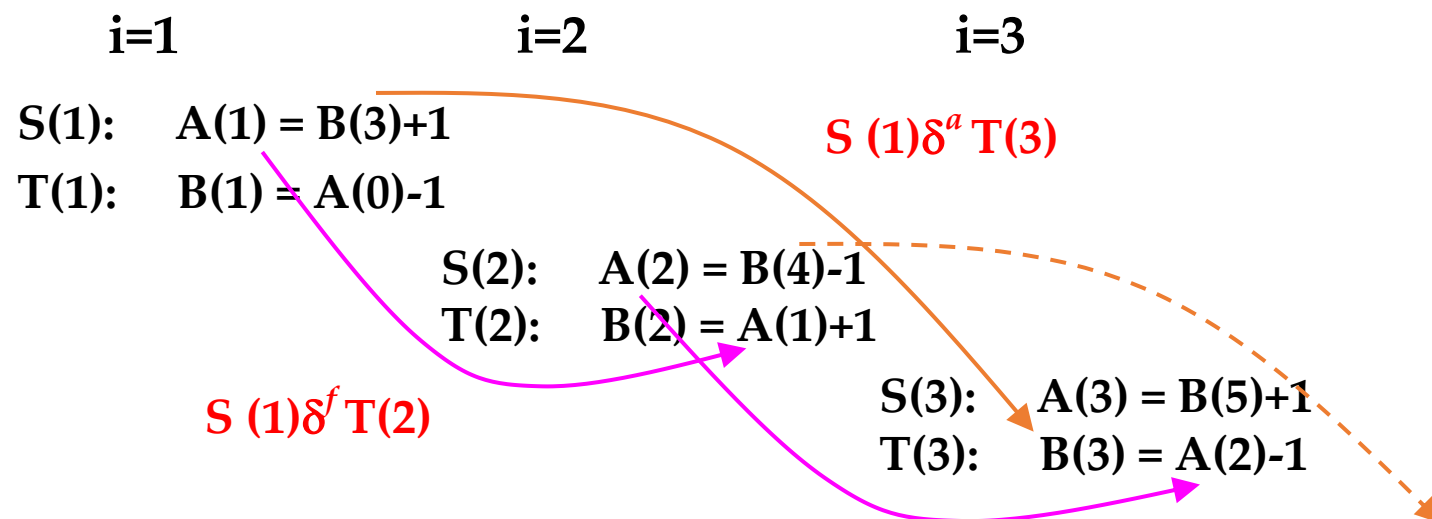
```
for (int m = 0; m < M; m++) {  
    for (int n = 0; n < N; n++) {  
        C[m][n] = 0;  
        for (int k = 0; k < K; k++) {  
            C[m][n] += A[m][k] * B[k][n];  
        }  
    }  
}
```

矩阵乘法

循环中的数据依赖：观察

```
do l=1,100
S:   A(l) = B(l+2)+1
T:   B(l) = A(l-1)-1
enddo
```

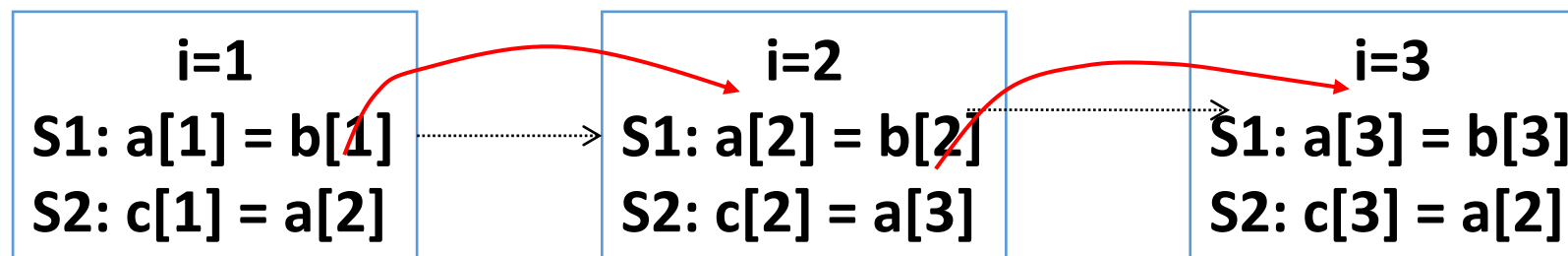
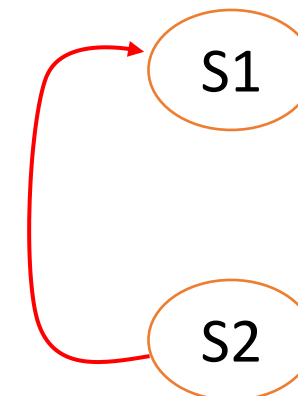
S(i)是S的一个实例
T(j)是T的一个实例



■ 循环语句 $\neq \triangleleft$

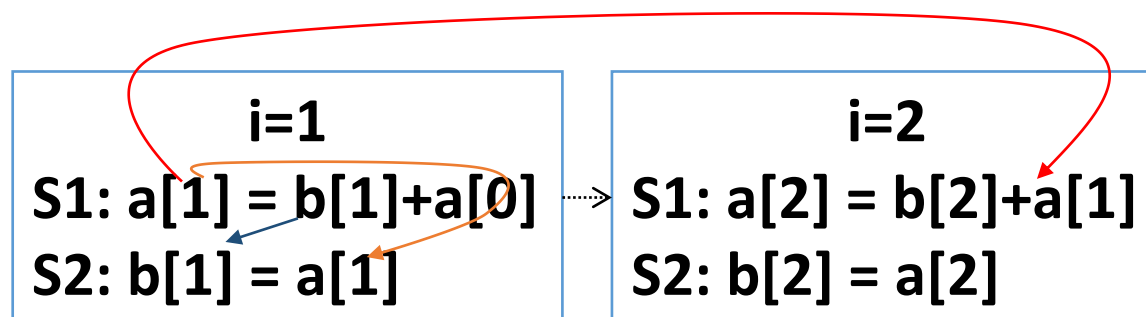
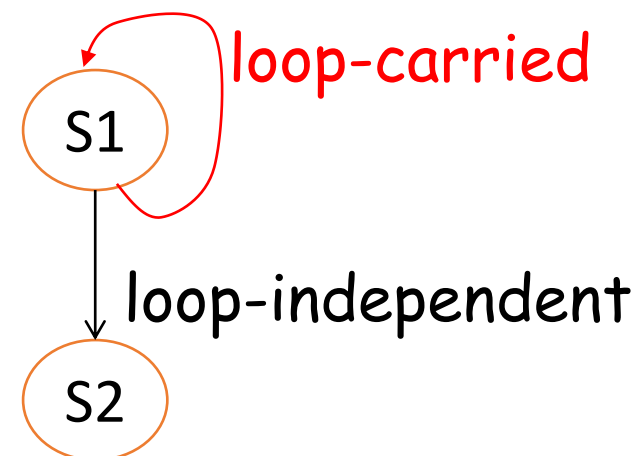
```
for (i=1; i<n; i++) {
  S1:  a[i] = b[i];
  S2:  c[i] = a[i+1];
}
```

loop-carried



■ 循环语句 \neq \triangleleft

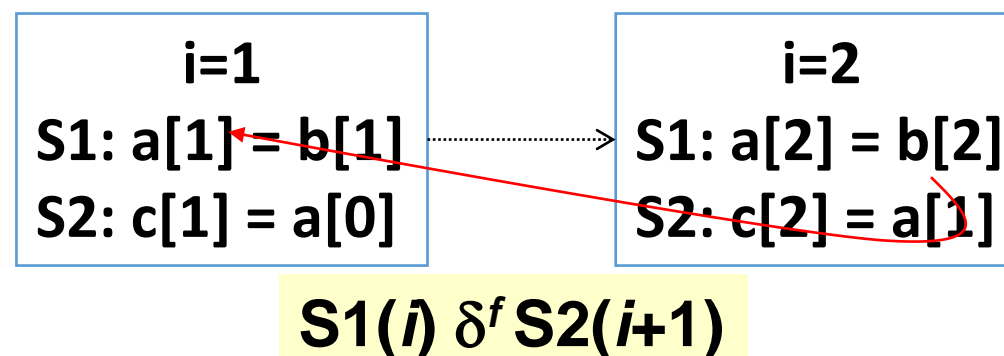
```
for (i=1; i<n; i++) {
  S1:  a[i] = b[i] + a[i-1];
  S2:  b[i] = a[i];
}
```



■ 嵌套循环L中语句T的一个实例T(J)依赖于语句S的一个实例S(I),
如果存在一个存储单元M, 且

- ⊕ S(I)和T(J)都引用(读或写)M, 至少一个是写
- ⊕ 在程序串行执行时, S(I)在T(J)之前执行
- ⊕ 在程序串行执行时, 从S(I)执行结束到T(J)开始执行前, 没有其他对M的写操作

```
for (i=1; i<n; i++) {
S1:  a[i] = b[i];
S2:  c[i] = a[i-1];
}
```



$S1(i) \delta^f S2(i+1)$

■ **T(J)流依赖于S(I)，如果S(I)写M而T(J)读M**

$$\oplus S(I) \delta^f T(J)$$

■ **T(J)反依赖于S(I)，如果S(I)读M而T(J)写M**

$$\oplus S(I) \delta^{-1} T(J)$$

■ **T(J)输出依赖于S(I)，如果S(I)和T(J)都写M**

$$\oplus S(I) \delta^o T(J)$$

■若 $i < j$, 循环携带的依赖 (loop-carried)

⊕跨迭代的依赖

```
for (i=1; i<n; i++)  
S:   a[i] = a[i-1];
```

 $S(i) \delta^f S(i+1)$

■若 $i = j$, 循环无关的依赖 (loop-independent)

⊕迭代内的依赖

```
for (i=1; i<n; i++) {  
S1:   a[i] = b[i];  
S2:   c[i] = a[i];  
}
```

 $S1(i) \delta^f S2(i)$

区分语句实例编号和数组下标

- 设语句S和T是嵌套循环L(嵌套深度=m)中的语句，如果语句T依赖于S，则存在实例 $S(i_1, i_2, \dots, i_m) \delta T(j_1, j_2, \dots, j_m)$
- 依赖距离向量 $d=(d_1, d_2, \dots, d_m)$ ， $d_k = j_k - i_k$
 - ⊕ 注意区分：语句实例编号、数组下标

```

    for (i=1; i<n; i++) {
S1:    a[i] = b[i];
S2:    c[i] = a[i-1];
    }

```

$S1(i) \delta^f S2(i+1)$

距离向量 $d=(1)$

- **依赖距离向量** $d=(d_1, d_2, \dots, d_m)$, $d_k = j_k - i_k$
- **依赖距离向量指明了对同一个存储单元的两次访问之间相隔的循环迭代数**
- **很多情况下，精确的依赖距离很难获取，仅能获取依赖方向和依赖层次**

■ **依赖方向向量** $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_m)$ $\sigma_k = \text{sign}(\text{依赖距离})$

$$\sigma_k = \begin{cases} < & \text{if } d_k > 0 \\ = & \text{if } d_k = 0 \\ > & \text{if } d_k < 0 \end{cases} \quad \sigma_k = \begin{cases} 1 & \text{if } d_k > 0 \\ 0 & \text{if } d_k = 0 \\ -1 & \text{if } d_k < 0 \end{cases}$$

■ **依赖方向向量指明了存在依赖关系的两个迭代在每一维上的依赖方向**

```
for (i=1; i<n; i++) {
  S1:  a[i] = b[i];
  S2:  c[i] = a[i-1];
}
```

$S1(i) \delta^f S2(i+1)$

依赖方向 $\sigma = "<"$ 或 $\sigma = "1"$

■ 依赖层次指明了是由哪一层循环引起的依赖关系

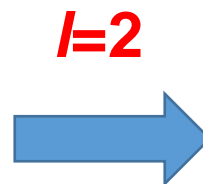
- ⊕ Level是依赖方向向量第一个非“=”的索引
- ⊕ 如果T在第l层上依赖于S, $1 \leq l \leq m$, 则称T对S的依赖是循环携带的依赖
- ⊕ $l=m+1$, 即依赖距离= $\{0,0,\dots,0\}$, 循环无关的依赖

■ 依赖层可以帮助编译器快速确定循环变换适用性

⊕ 如果依赖层为 l ，那么在 $l-1$ 层都可以进行循环并行化

```

for (i=1; i<n; i++)
  for(j=1;j<n;j++)
  {
S1:    a[i,j] = b[i,j];
S2:    c[i,j] = a[i,j-1];
  }
    
```

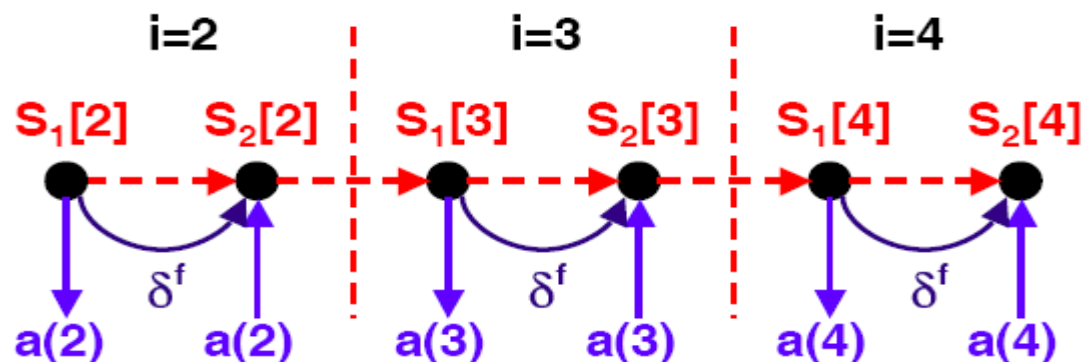


```

#pragma omp parallel for
for (i=1; i<n; i++)
  for(j=1;j<n;j++)
  {
S1:    a[i,j] = b[i,j];
S2:    c[i,j] = a[i,j-1];
  }
    
```

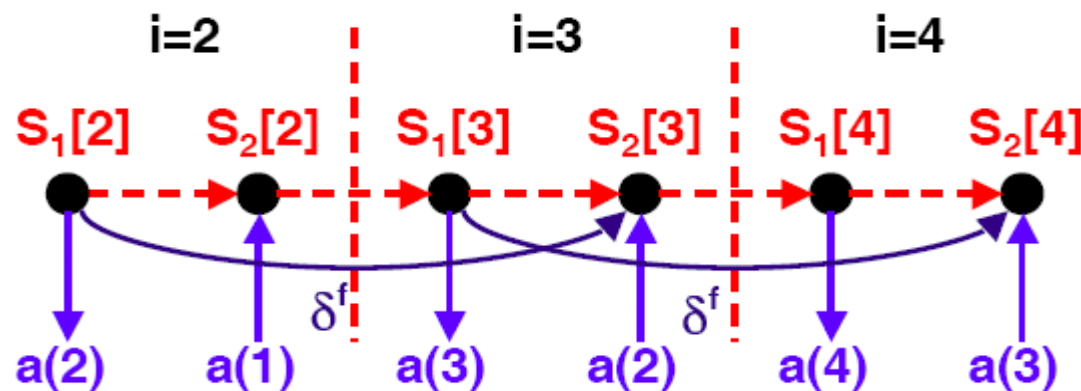
例子：循环无关的数据依赖

```
for(i= 2;i <=4; i++)
{
  S1: a(i)= b(i) + c(i)
  S2: d(i) = a(i)
}
```



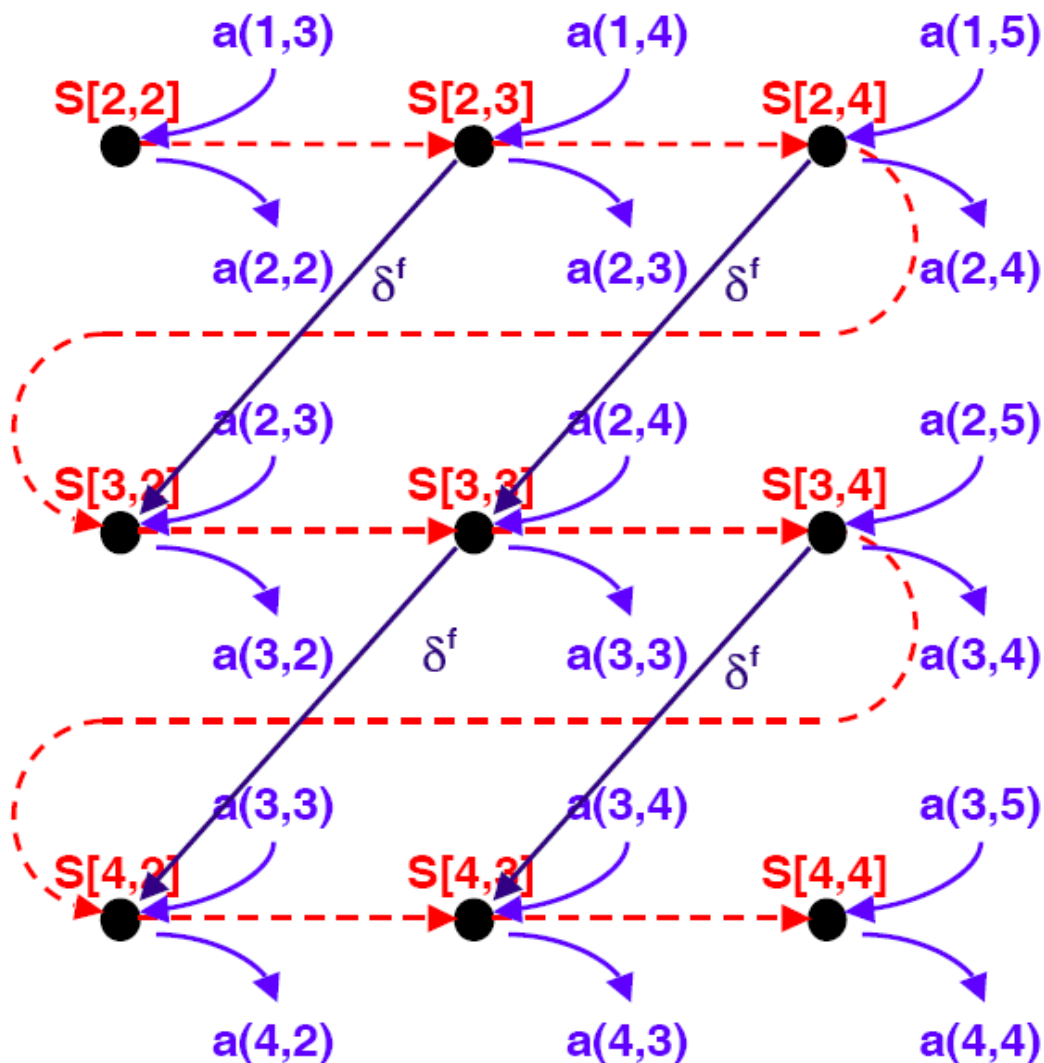
- 存在语句S1的一个实例在语句S2的实例之前执行，且S1生产数据、S2消费数据，则S1是数据依赖的源，S2是数据依赖的槽
- 依赖距离为 0，依赖方向为 =
- 数据依赖表示： $S_1 \delta_0^f S_2$ $S_1 \delta_{=}^f S_2$

```
for(i= 2;i <=4; i++)
{
  S1: a(i)= b(i) + c(i)
  S2: d(i) = a(i-1)
}
```



- 存在语句S1的一个实例在语句S2的实例之前执行，且S1生产数据、S2消费数据，则S1是数据依赖的源，S2是数据依赖的槽
- 依赖距离为 1，依赖方向为 正(<)
- 数据依赖表示： $S_1 \delta_1^f S_2$ $S_1 \delta_{<}^f S_2$

例子：循环携带的依赖关系



```
for(i= 2;i <=4; i++)
  for (j=2; j<=4; j++)
    S:  a(i, j)= a(i-1, j+1)
```

■ S是依赖源也是依赖槽

■ 依赖距离是(1, -1)

■ 数据依赖表示

$$S \delta_{(<, >)}^f S \quad S \delta_{(1, -1)}^f S$$

- **目标：用静态方法识别数据依赖关系**
- **为了获得嵌套循环L的依赖关系信息，需要找出循环体中每一对变量所引起的数据依赖**
- **基本的依赖关系问题是判断循环L中含索引变量的两个下标确定的同名数组元素在给定条件下是否表示同一个存储单元**
 - ⊕ **大多数数组元素引用的下标均是循环索引变量的线性表达式，因此主要考虑线性下标数组元素引用导致的依赖**

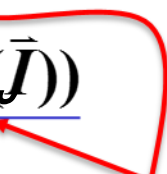
■考虑下面嵌套深度为m的循环L

```

do  $i_1 = l_1, u_1, s_1$ 
  do  $i_2 = l_2, u_2, s_2$ 
    ...
    do  $i_m = l_m, u_m, s_m$ 
      S1     $A(f_1(\vec{I}), f_2(\vec{I}), \dots, f_n(\vec{I})) = \dots$ 
      S2     $\dots = A(g_1(\vec{J}), g_2(\vec{J}), \dots, g_n(\vec{J}))$ 
    enddo
    ...
  enddo
enddo

```

linear function



存在数据依赖仅当存在两个循环索引向量

$$\vec{i}, \vec{j} \text{ such that } \vec{L} \leq \vec{i}, \vec{j} \leq \vec{U}$$

$$\text{and } f_k(\vec{i}) = g_k(\vec{j}), 1 \leq k \leq n$$

■ 依赖关系测试等价于整数线性规划问题

$$(1) \quad \begin{cases} f_1(\vec{i}) - g_1(\vec{j}) = 0 \\ f_2(\vec{i}) - g_2(\vec{j}) = 0 \\ \dots \\ f_n(\vec{i}) - g_n(\vec{j}) = 0 \end{cases} \quad \begin{aligned} \vec{i} &= (i_1, i_2, \dots, i_m) \\ \vec{j} &= (i'_1, i'_2, \dots, i'_m) \end{aligned}$$

$$(2) \quad \vec{L} \leq \vec{i}, \vec{j} \leq \vec{U}$$

■ 依赖距离向量 $\vec{j} - \vec{i}$

■ 依赖方向向量 $sign(\vec{j} - \vec{i})$

- **第一目标：证明对同一数组元素的两次访问间没有依赖**
 - ⊕ 通过证明方程 (1) - (2) 没有解
- **次要目标：试图刻画可能的依赖关系**
 - ⊕ 依赖距离向量、方向向量的最小完整集合
- **依赖关系测试是保守的 (Conservative)**
 - ⊕ 不能反驳依赖关系存在时只能假设 “存在依赖”

Practical Dependence Testing *

Gina Goff Ken Kennedy Chau-Wen Tseng

Department of Computer Science
Rice University
Houston, TX 77251-1592



- ← 1967 Bachelor@Rice U.
- ← 1969 M.S. @NY U.
- ← 1971 Ph.D. @NY U.
- ← 1971 Back to Rice U.

1979-80 On sabbatical at IBM Research at Yorktown Heights

- ← 1980 Full Prof. Rice U.
- ← 1984 found CS of Rice U.

1980-97 **Vectorization** in PFC and IBM Fortran Compiler, and later **parallelization** and **memory hierarchy** (15 PhD)

- ← 1995 Fellow of ACM & IEEE
- ← Until 2007

OPTIMIZING COMPILERS
for
MODERN ARCHITECTURES
Randy Allen & Ken Kennedy



ACM and IEEE CS co-sponsor the Kennedy Award, which was established in 2009 to recognize substantial contributions to programmability and productivity in computing and significant community service or mentoring contributions. It was named for the late Ken Kennedy, founder of Rice University's computer science program and a world expert on high performance computing. The Kennedy Award carries a US \$5,000 honorarium endowed by IEEE CS and ACM. The award will be formally presented to Abramson in

dependence analysis, making it more practical for all compilers. We begin with some definitions.

1.1 Data Dependence

The theory of *data dependence*, originally developed for automatic vectorizers, has proved applicable to a wide range of optimization problems. We say that a data dependence exists between two statements S_1 and S_2 if there is a path from S_1 to S_2 and both statements access the same location in memory. There are four types of data dependence [32, 33]:

True (flow) dependence occurs when S_1 writes a memory location that S_2 later reads.

Anti dependence occurs when S_1 reads a memory location that S_2 later writes.

Output dependence occurs when S_1 writes a memory location that S_2 later writes.

Input dependence occurs when S_1 reads a memory location that S_2 later reads.

Dependence analysis is the process of computing all such dependences in a program.

1.2 Dependence Testing

Calculating data dependence for arrays is complicated by the fact that two array references may not always access the same memory location. *Dependence testing* is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest. For the purposes of this explication, we will ignore any control flow except for the loops themselves. Suppose that we wish to test whether or not there exists a dependence from statement S_1 to S_2 in the following model loop nest:

```
DO  $i_1 = L_1, U_1$ 
  DO  $i_2 = L_2, U_2$ 
    ...
    DO  $i_n = L_n, U_n$ 
       $S_1$   $A(f_1(i_1, \dots, i_n), \dots, f_m(i_1, \dots, i_n)) = \dots$ 
       $S_2$   $\dots = A(g_1(i_1, \dots, i_n), \dots, g_m(i_1, \dots, i_n))$ 
      ENDDO
    ...
  ENDDO
ENDDO
```

Let α and β be vectors of n integer indices within the ranges of the upper and lower bounds of the n loops in the example. There is a dependence from S_1 to S_2 if and only if there exist α and β such that α is lexi-

Abstract

Precise and efficient dependence tests are essential to the effectiveness of a parallelizing compiler. This paper proposes a dependence testing scheme based on classifying pairs of subscripted variable references. Exact yet fast dependence tests are presented for certain classes of array references, as well as empirical results showing that these references dominate scientific Fortran codes. These dependence tests are being implemented at Rice University in both PFC, a parallelizing compiler, and ParaScope, a parallel programming environment.

1 Introduction

In the past decade, high performance computing has become vital for scientists and engineers alike. Much progress has been made in developing large-scale parallel architectures composed of powerful commodity microprocessors. To exploit parallelism and the memory hierarchy effectively for these machines, compilers must be able to analyze data dependences precisely for array references in loop nests. Even for a single microprocessor, optimizations utilizing dependence information can result in integer factor speedups for scientific codes [11]. However, because of its expense, few if any scalar compilers perform dependence analysis.

Parallelizing compilers have traditionally relied on two dependence tests to detect data dependences between pairs of array references: Banerjee's inequalities and the GCD test [8, 55]. However, these tests are usually more general than necessary. This paper presents empirical results showing that most array references in scientific Fortran programs are fairly simple. For these simple references, we demonstrate a suite of highly exact yet efficient dependence tests. We feel that these tests will significantly reduce the cost of performing

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, by IBM Corporation, and by the Cray Research Foundation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-428-7/91/0005/0015...\$1.50

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28, 1991.

■ 根据索引变量在数组下标中的出现情况分类

⊕ ZIV: Zero index variable 51%

⊕ SIV: Single index variable 46%

⊕ MIV: Multiple index variable 3%

■ 依赖测试复杂度与数组下标的复杂性相关

⊕ 下例三个下标分别是哪种情况？

```
DO i
  DO j
    DO k
      S1          A(5, i+1, j) = A(N, i, k) + C
    ENDDO
  ENDDO
ENDDO
```

■根据索引变量在数组下标间是否共享

- ⊕可分的 (Separable) : 每个下标表达式的索引变量不同
- ⊕对偶的 (Coupled) : 下标表达式共享索引变量, 不同下标包含相同的索引变量

```
DO i
  DO j
    DO k
      S1      A(i, j, j) = A(i, j, k) + C
    ENDDO
  ENDDO
ENDDO
```

第一个下标是可分的, 第二、三个是对偶的

■ 利用可分属性独立地测试数据依赖

Separable subscripts may be tested independently, and merge the resulting dependence information.

```
      DO i
        DO j
          DO k
S1           A(i+1, j, k-1) = A(i, j, k) + C
              ENDDO
            ENDO
          ENDDO
```

依赖距离向量 (1, 0, -1)

■ Partition-Based Algorithm

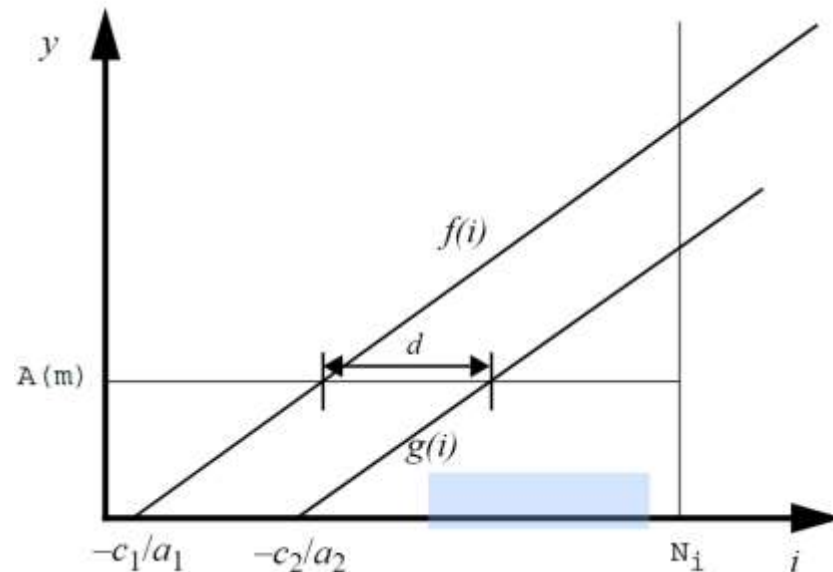
1. Partition into separable & coupled groups
2. Classify as ZIV, SIV, MIV subscripts
3. Apply dependence tests to each group
4. Finished if any test yields independence
5. Otherwise, merge dependence information

- $A(e_1)$ and $A(e_2)$
- 如果 $e_1 \neq e_2$, 那么两次访问是独立的

■ Test $A(a_1 * i + c_1)$ and $A(a_2 * i + c_2)$

■ Strong SIV when $a_1 = a_2$

■ Lamport测试法



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

■ Lamport测试法的适应场景

- ⊕ 用于下标表达式只包含一个索引变量，并且两个表达式的索引变量的因子相同的情况

$$A(\cdots, b * i + c_1, \cdots) = \cdots$$

$$\cdots = A(\cdots, b * i + c_2, \cdots)$$

■ 依赖问题转换为是否存在 i_1 和 i_2 满足

$$b * i_1 + c_1 = b * i_2 + c_2$$

$$L_i \leq i_1, i_2 \leq U_i$$

■ 上面的方程有整数解 当且仅当

$$\frac{c_1 - c_2}{b} \text{ 是一个整数}$$

■ 依赖距离为 $d = \frac{c_1 - c_2}{b}$, if $|d| \leq U_i - L_i$

⊕ $d > 0 \Rightarrow$ 流依赖

⊕ $d = 0 \Rightarrow$ 循环无关依赖

⊕ $d < 0 \Rightarrow$ 反向依赖

Lamport测试法

```
do i = 1, n
  do j = 1, n
    S: a(i, j) = a(i-1, j+1)
  end do
end do
```

$$\blacksquare i_1 = i_2 - 1$$

$$\blacksquare b = 1; c_1 = 0; c_2 = -1$$

$$\blacksquare L_i = 1, U_i = n$$

There is dependence

$$\text{Distance}(i) = 1$$

$$\blacksquare j_1 = j_2 + 1$$

$$\blacksquare b = 1; c_1 = 0; c_2 = 1$$

$$\blacksquare L_j = 1, U_j = n$$

There is dependence

$$\text{Distance}(j) = -1$$

$$S\delta_{(1,-1)}^f S \quad \text{or} \quad S\delta_{(<,>)}^f S$$

■ Weak-0 SIV when $a_1 = 0$ or $a_2 = 0$

$$i = (c_2 - c_1) / a_1$$

⊕ Check that i is an integer and within the loop bounds.

⊕ Weak-0 SIV通常出现在循环头尾迭代

⊕ 利用循环剥离可以消除这种依赖

```
DO  $i = 1, n$   
S:    $y(i, n) = y(1, n) + y(n, n)$   
ENDDO
```

$i = 1$ or $i = n$ 有依赖

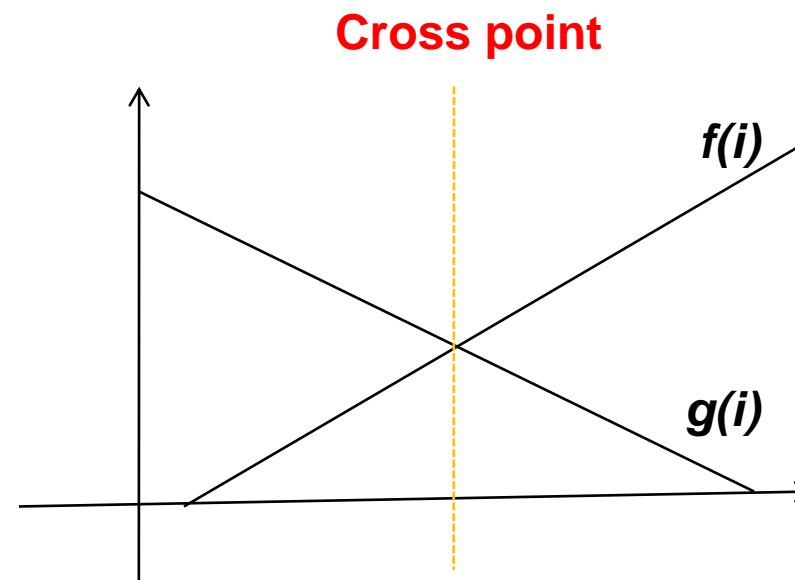
■ Weak Crossing SIV when $a_1 = -a_2$

$$i = (c_1 - c_2) / (2 * a_1)$$

⊕ Check that the resulting value i is within the loop bounds, and is either an integer or has a non-integer part equal to $1/2$.

⊕ 利用循环分裂可以消除这种依赖

```
DO  $i = 1, n$   
S:    $y(i) = y(n-i+1)$   
ENDDO
```



■ Multiple Induction Variable Tests

⊕ GCD测试法

⊕ Banerjee不等式测试法

⊕ In-Exact

$$\begin{cases} f_1(\vec{i}) - g_1(\vec{j}) = 0 \\ f_2(\vec{i}) - g_2(\vec{j}) = 0 \\ \dots \\ f_n(\vec{i}) - g_n(\vec{j}) = 0 \end{cases} \quad \begin{aligned} \vec{i} &= (i_1, i_2, \dots, i_m) \\ \vec{j} &= (i'_1, i'_2, \dots, i'_m) \end{aligned}$$

$$a_0 + a_1x_1 + \dots + a_nx_n - (b_0 + b_1y_1 + \dots + b_ny_n) = 0 \quad (1)$$

$$a_1x_1 - b_1y_1 + \dots + a_nx_n - b_ny_n = b_0 - a_0$$

$$\sum_{i=1}^n a_i x_i = c, \text{ } a_i \text{ and } c \text{ are integers} \quad (*)$$

线性丢番图方程

■ 对于给定方程 $\sum_{i=1}^n a_i x_i = c$, a_i and c are integers

存在整数解 当且仅当

$\gcd(a_1, a_2, \dots, a_n)$ 被 c 整除

■ 讨论

⊕ 如果条件不成立，方程无解，则无依赖存在

⊕ 如果条件成立，一定有依赖吗？ $\bar{L} \leq \bar{i}, \bar{j} \leq \bar{U}$

```
do i = 1, 10
  S1: a(2*i)=b(i)+c(i)
  S2: d(i) = a(2*i-1)
end do
```

- 是否存在两个迭代向量 i_1 和 i_2 , 满足
- 存在一个整数解 当且仅当

$$\begin{cases} 1 \leq i_1, i_2 \leq 10 \\ 2 * i_1 = 2 * i_2 - 1 \end{cases}$$

$\gcd(2, -2) = 2$ 整除 -1?

- No! **不存在**依赖关系!
- 使用Lamport测试法

S1(1): store a(2)
S2(1): load a(1)

S1(2): store a(4)
S2(2): load a(3)

...

```
do i = 1, n
  do j = 1, n
    S: a(2*i+1,2*j-3) = a(i+j,2*j)
  end do
end do
```

$$\blacksquare 2*i_1 + 1 = i_2 + j_2$$

$$\blacksquare \gcd(2, -1, -1) \text{ divides } -1?$$

$$\blacksquare \text{yes}$$

$$\blacksquare 2*j_1 - 3 = 2*j_2$$

$$\blacksquare \gcd(2, -2) \text{ divides } 3?$$

$$\blacksquare \text{no}$$

没有依赖

- $i_1 = i_2 + 100$
- gcd(1,-1) 整除 100?
- **yes**
- 存在依赖关系!?

```
do i = 1, 10
  S1: a(i) = b(i) + c(i)
  S2: d(i) = a(i+100)
end do
```

S1: a(1)	a(2)	...	a(10)
S2: a(101)	a(102)	...	a(110)

没有依赖

■ GCD测试法没有考虑循环上下界

⊕ 无法考虑方程 $\bar{L} \leq \bar{i}, \bar{j} \leq \bar{U}$ (2)

■ GCD没有给出依赖距离或者方向的任何信息

■ 系数的最大公约数经常为1 ($\gcd(\dots) = 1$) , 此时可以整除c, 导致产生假依赖

■ 1. 数据依赖关系定义

⊕ 数据依赖的基本定义、分类、表示方法

■ 2. 循环中的数据依赖关系

⊕ 循环中的数据依赖

■ 3. 依赖距离、方向、层次

⊕ 与数据依赖相关的三个基本概念

■ 4. 数据依赖举例

⊕ 给定代码，如何表示数据依赖关系

■ 依赖关系测试器

- ⊕ 用于确定是否存在两个循环索引向量 \vec{i}, \vec{j} 满足所述两个约束的算法

- ⊕ 精确测试法与近似测试法

■ 依赖关系测试必须是保守的

- ⊕ 两个独立的访问常常不能被证明

■ 依赖关系测试是NP完全问题

- ⊕ 求解线性丢番图

■ 依赖关系测试

Testing Algorithm	
Lamport's Test	Exact
GCD Test	In-exact
Banerjee's Inequalities	In-exact
Generalized GCD Test	In-exact, 处理非紧嵌套循环
λ -Test	Banerjee's Inequalities的多维形式
Delta Test	λ -Test的受限制形式
Power Test	基于Fourier-Motzkin消去法将循环边界应用到从多维GCD测试得到的稠密系统上，精确的信息
Omega Test	基于Fourier-Motzkin消去法对整数规划的扩展（GCC使用）

■ 对于给定代码，分析存在的数据依赖关系，并指出依赖的源和槽、依赖类型、依赖方向

- ⊕ 使用循环展开的方法分析数据依赖
- ⊕ 使用Lamport测试法分析数据依赖

```
for(i= 2;i <=4; i++)  
    for (j=2; j<=4; j++)  
S:      a(j, i)= a(j+1, i-1)
```

- **《高级编译器设计与实现》 (鲸书) 第9章**
- **Building an Optimizing Compiler, ch3.6**
- **Optimizing Compilers for Modern Architectures, ch2-ch3**
- **参考论文**
 - ⊕ **Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck: An Efficient Method of Computing Static Single Assignment Form. POPL 1989: 25-35**
 - ⊕ **A practical algorithm for exact array dependence analysis(Omega)**
 - ⊕ **Efficient and exact data dependence analysis(SUIF)**
 - ⊕ **Gina Goff, Ken Kennedy, Chau-Wen Tseng: Practical Dependence Testing. PLDI 1991: 15-29**