并行计算
Parallel Computing

主讲 孙经纬

2024年 春季学期

概要

- 第三篇 并行编程
  - 第十三章 并行程序设计基础
  - 第十四章 共享存储系统并行编程
  - 第十五章 分布存储系统并行编程
  - 补充章节1 GPU并行编程
  - **补充章节2 关于并行编程的更多话题**

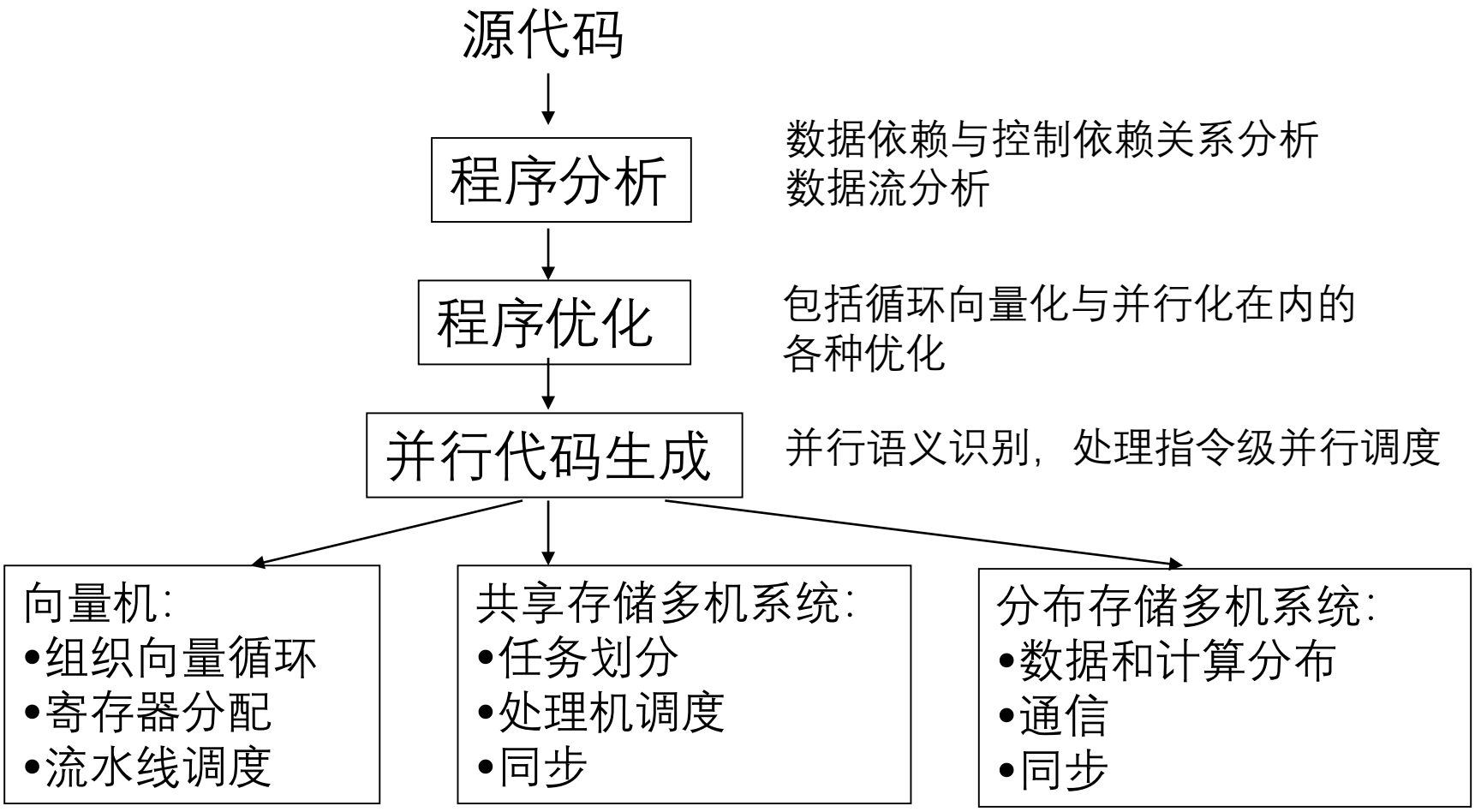
关于并行编程的更多话题

- 数据依赖与自动并行
- MapReduce
- 并行程序性能优化
- 案例：图与矩阵算法的GPU优化

数据依赖与自动并行

- **并行编译器**
- 数据依赖关系
- 循环的向量化与并行化

并行编译器的组成及任务

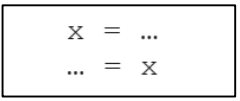


数据依赖与自动并行

- 并行编译器
- **数据依赖关系**
- 循环的向量化与并行化

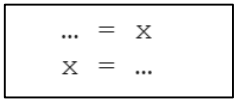
数据依赖关系

- Def1：语句S和T，若存在变量x使之满足下述条件之一，则称语句T依赖于语句S，记为S δ T，否则S和T之间没有数据依赖关系：



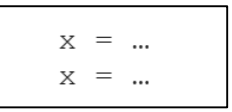
(1) 流依赖： $S\ \delta^f\ T$，若x∈OUT（S）且 x∈IN（T）

T使用S计算出的x的值：T流依赖于S;



(2) 反依赖： $S\ \delta^a\ T$，若x∈IN（S）且 x∈OUT（T）

S使用x值先于T对x的定值：T反依赖于S;



(3) 输出依赖：$S\ \delta^o\ T$，若x∈OUT（S）且x∈OUT（T）

S较之T先对x进行定值：T输出依赖于S;

依赖关系示例

- e.g. 考虑语句序列:

S : A = B + D IN: B D , OUT: A

T : C = A * 3 IN: A , OUT: C

U : A = A + C IN: A C , OUT: A

V : E = A / 2 IN: A , OUT: E

S δ^f T S δ^f U S δ^o U T δ^f U T δ^a U U δ^f V

依赖关系示例

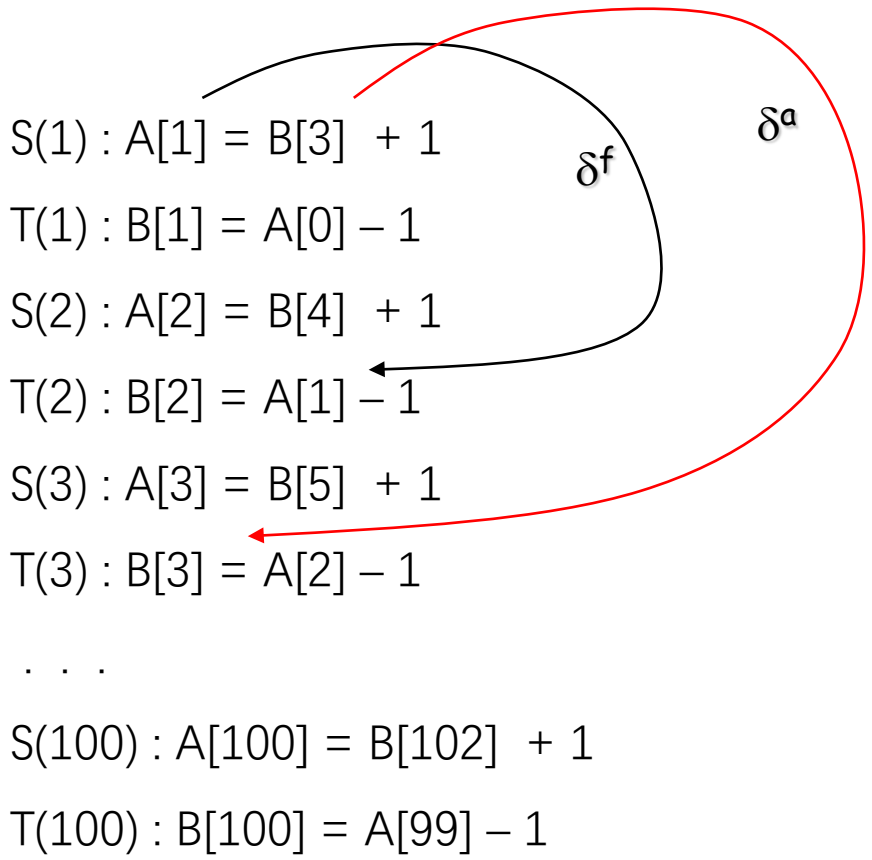
e.g. 循环语句:

```
    for i = 1 to 100 do
S :     A[i] = B[ i+2] + 1;
T :     B[i] = A[i-1] – 1;
    end for
```

依赖关系:

S δ^f T

S δ^a T



数据依赖关系

- Def2：语句S和T在循环L中。如果S的实例S(i)和T的实例T(j)以及变量u∈S，变量v ∈T，满足：

  （1）u和v至少有一个是输出变量；

  （2）u∈S(i)和变量v ∈T(j)***表示同一个存储单元M***

  （3）在L的顺序执行中，S(i)先于T(j)

  （4）在L的顺序执行中，S(i)之间T(j)没有其他对M的写操作；

  则u、v引起T依赖于S，即S δ T，称为T(j)依赖于S(i)，其中：

  流依赖：　u ∈OUT(S) , v ∈IN(T)

  反依赖：　u ∈IN(S) ，　v ∈OUT(T)

  输出依赖：u ∈OUT(S) , v ∈OUT(T)

T 对S的依赖即为满足上述条件的偶对（S(i),T(j)）的集合。

依赖距离和依赖向量

令**α**=($\alpha_1,\alpha_2,\cdots,\alpha_n$) 和**β**=($\beta_1,\beta_2,\cdots,\beta_n$)是m层循环内的n个整数下标向量，假定**α**和**β**存在数据相关性，则

- **依赖距离向量**（Dependent Distance Vector）

  D = ($D_1,D_2,\cdots,D_n$)定义为**β-α**;

- **依赖方向向量**（Dependent Direction Vector）

  d = ($d_1,d_2,\cdots,d_n$)定义为：

$$d_i = \begin{cases} < & \alpha_i < \beta_i \\ = & \alpha_i = \beta_i \\ > & \alpha_i > \beta_i \end{cases}$$

也可用 1表示<

也可用 0表示=

也可用 -1表示>

依赖距离和依赖向量

例如，有如下的三层循环嵌套：

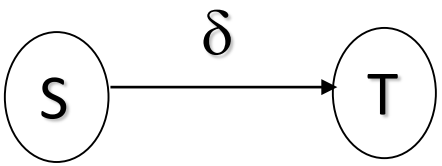
for i = l_1 **to** u_1 **do**
  for j = l_2 **to** u_2 **do**
    for k = l_3 **to** u_3 **do**
      A(i + 1, j, k – 1) = A(i, j, k) + C
    endfor
  endfor
endfor

则数组A的三维迭代之间的依赖距离向量D = (i + 1 – i, j – j, k – 1 – k) = (1, 0, -1)和依赖方向向量= (<, =, >)。

依赖方向向量对计算循环体间依赖关系十分有用，其依赖关系是通过依赖方向向量不是"="号的外层循环传递的；依赖距离向量指明在同一存储单元的两次访问之间循环迭代的实际距离。它们对开发并行性或优化存储器层次结构时起到指引作用。

语句依赖图和迭代依赖图

- 语句依赖图 – 依赖关系δ的有向图。将语句，如S和T，看成节点，若有S δ T，则:



间接依赖关系： $\bar{\delta}$ ，即依赖关系δ的传递闭包。

若S $\bar{\delta}$ T，则在语句依赖图中存在S到T的一条路径。

- 迭代依赖图 – 即（循环）迭代之间的依赖关系。在循环L中，若语句T依赖于语句S，即S δ T。令S(i)和T(j)是满足依赖关系的偶对，S(i) δ T(j)，此时应该有i≤j。在i<j时，称迭代H(j)依赖于迭代H(i)，记为H(i) δ H(j)。

语句依赖图示例

有如下循环语句:

```
for i = 4 to 200 do
    S:    A(i) = B(i) + C(i)
    T:    B(i+2) = A(i-1) + A(i-3) + C(i-1)
    U:    A(i+1) = B(2*i+3) + 1
endfor
```

各语句间依赖关系如何呢?

- S δ^f T, T δ^f S
- U δ^o S
- U δ^a T

语句依赖图示例

- 语句T流依赖于语句S，即S δ^f T，满足依赖关系的偶对集合为：

{ <S(i), T(j)> | i = j - 1； 5≤j≤200 } ∪

{ <S(i), T(j)> | i = j - 3； 7≤j≤200 }

- 语句S流依赖于语句T，即T δ^f S，满足依赖关系的偶对集合为：**{ <T(i), S(j)> | i = j -2； 6≤j≤200 }**
- 语句S输出依赖于语句U，即 U δ^o S，满足依赖关系的偶对集合为：**{ <U(i), S(j)> | i = j -1； 5≤j≤200 }**
- 语句T反依赖于语句U，即U δ^a T，满足依赖关系的偶对集合为：**{ <U(i), T(j)> | j = 2*i + 1； 4≤i≤99 }**
- **语句T是否流依赖于语句U呢？**

语句依赖图示例

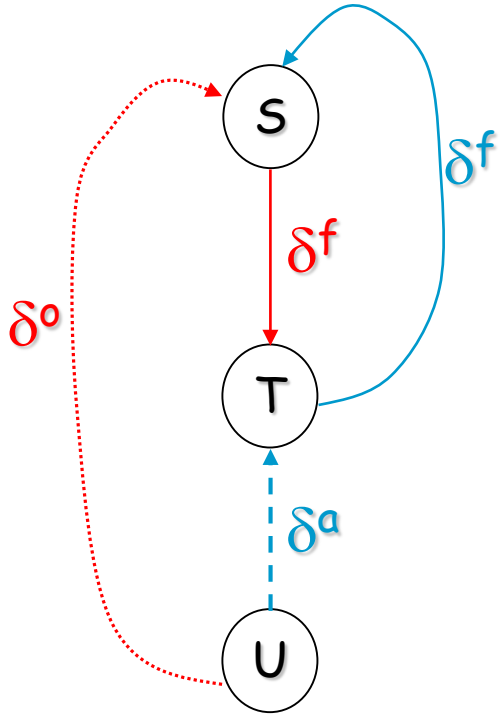
for i = 4 to 200 do

S: A(i) = B(i) + C(i)

T: B(i+2) = A(i-1) + A(i-3) + C(i-1)

U: A(i+1) = B(2*i+3) + 1

endfor



语句T是否流依赖于语句U呢?

迭代依赖图示例1

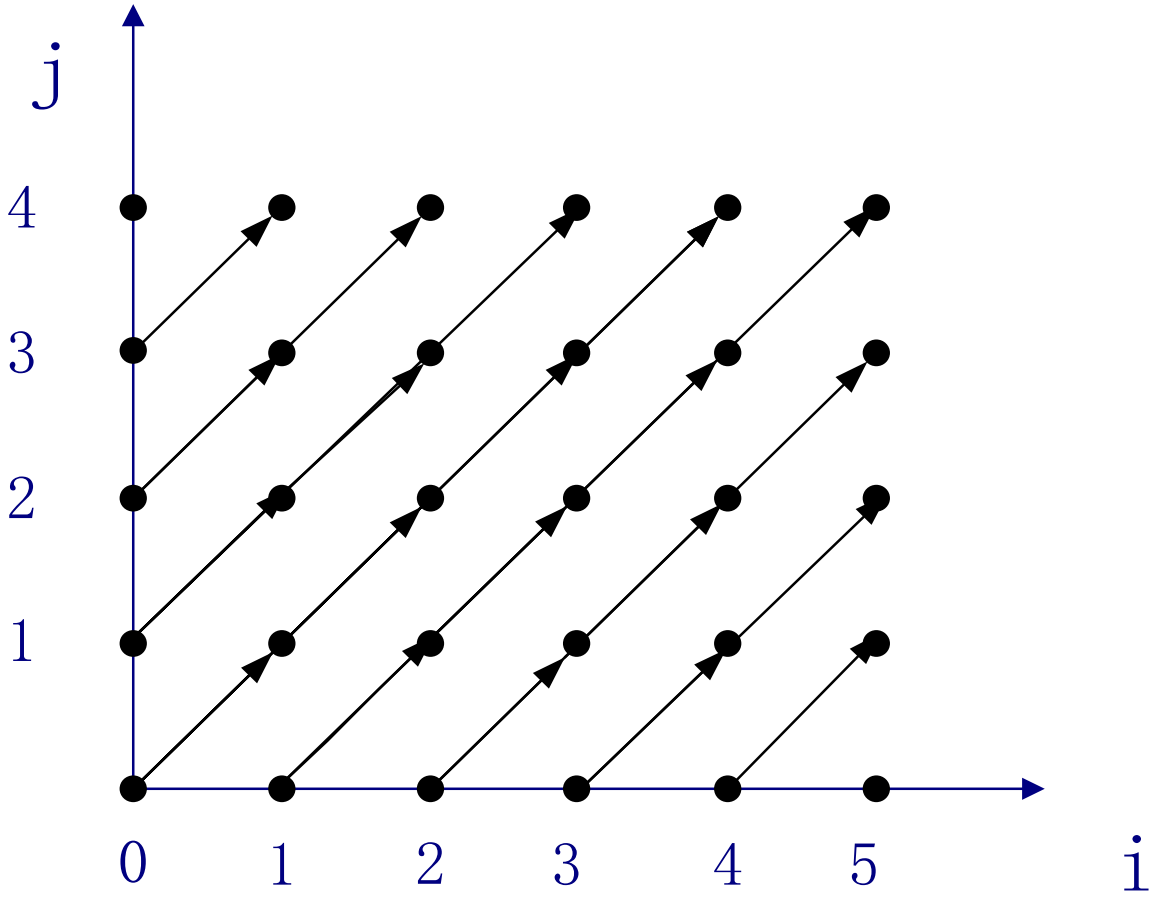
有如下二重循环:

```
for i = 0 to 5 do
   for j = 0 to 4 do
    S:      A(i+1, j+1) = A(i,j) + B (i,j)
    endfor
endfor
```

显然，$S\ \delta^f\ S$ 。满足依赖关系的偶对集合:

$\{ <S(i_1,i_2), S(j_1,j_2)> \mid j_1 = i_1 + 1 , j_2 = i_2 + 1,$
$0 \leqslant i_1 \leqslant 4, 0 \leqslant i_2 \leqslant 3$
$\}$ // 依赖方向向量和距离向量各是什么?

迭代依赖图示例1



迭代依赖图示例2

有如下二重循环:

```
L1 :    for i1 = 0 to 4 do
L2 :      for i2 = 0 to 4  do
  S :       A(i1+1, i2) = B(i1, i2) + C(i1, i2)
  T :       B(i1, i2+1) = A(i1, i2+1) + 1
  U :       D(i1, i2) = B(i1, i2+1) – 2
          endfor
        endfor
```

迭代依赖图示例2

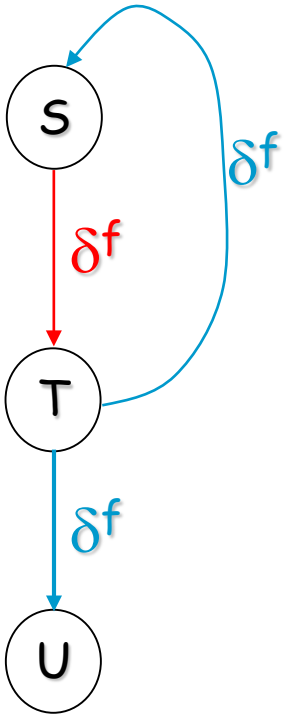
- 语句T流依赖于语句S，即S δ^f T，满足依赖关系的偶对：
  { < S(i1,i2), T(j1,j2) | j1 = i1+1, j2=i2-1,

  0≤i1≤3, 1≤i2≤4} ，距离向量为(1,-1)，方向向量为(1, -1)。此依赖关系由循环L1携带；
- 语句S流依赖于语句T，即T δ^f S，满足依赖关系的偶对：
  { < T(i1,i2), S(j1,j2) | j1 = i1, j2=i2+1,

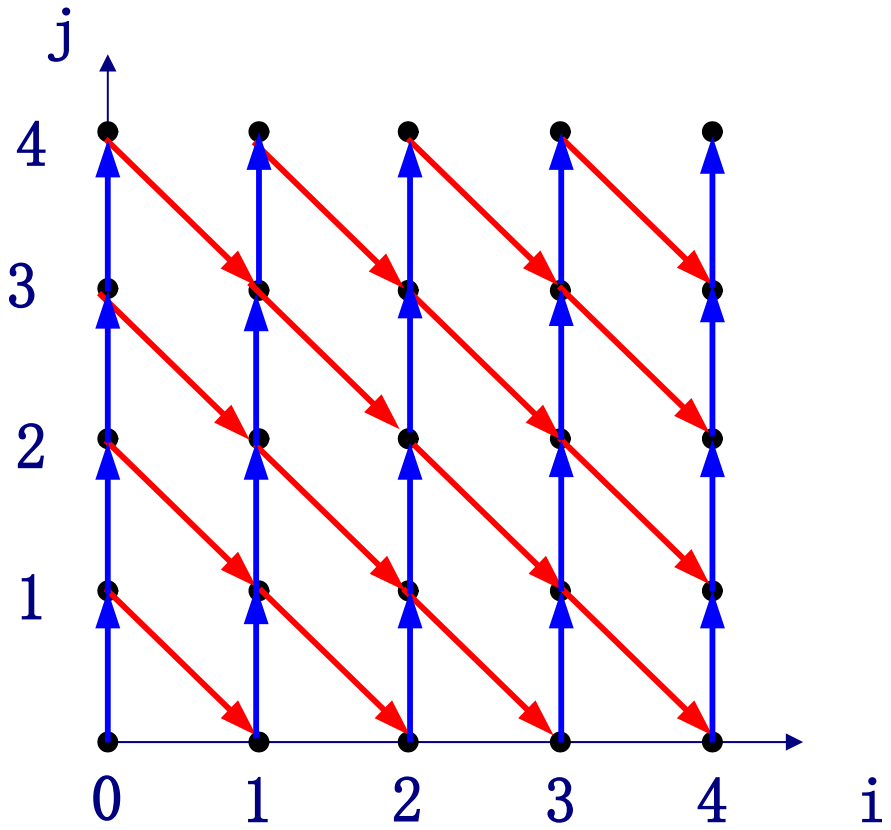
  0≤i1≤4, 0≤i2≤3} ，距离向量为(0,1)，方向向量为(0, 1)。此依赖关系由循环L2携带；
- 语句U流依赖于语句T，即T δ^f U，满足依赖关系的偶对：
  { < T(i1,i2), U(j1,j2) | j1 = i1, j2=i2,

  0≤i1≤4, 0≤i2≤4} ，距离向量为(0,0)，方向向量为(0,0)。此依赖关系与循环无关。

迭代依赖图示例2



语句依赖图



迭代依赖图

依赖关系方程

- 假定循环中数组下标是循环索引变量的线性表达式
- 循环的一般表示：(X是n维数组，f和g是循环索引变量$I=(I_1,I_2,\cdots,I_m)$ 的线性函数)

L_1: for $I_1 = p_1$ to q_1 do
L_2: for $I_2 = p_2$ to q_2 do
. . .
L_m: for $I_m = p_m$ to q_m
. . .
S : $X(f_1(I), f_2(I), \cdots, f_n(I)) =$ . . .
T : . . . = . . . $X(g_1(I), g_2(I), \cdots, g_n(I))$ . . .
. . .
endfor
. . .
endfor
endfor

依赖关系方程

- 上述循环L中语句S和T，令u= $X(f_1(I), f_2(I), \cdots, f_n(I))$ 是S的变量，而v= $X(g_1(I), g_2(I), \cdots, g_n(I))$ ， u或v至少一个是相应语句的输出变量。u、v导致S和T之间存在依赖关系，如果以下方程组

$$\begin{cases} f_1(I) - g_1(J) = 0 \\ f_2(I) - g_2(J) = 0 \\ \cdots \\ f_n(I) - g_n(J) = 0 \end{cases}$$

有满足约束条件$p_r \leqslant i_r \leqslant q_r$ $p_r \leqslant j_r \leqslant q_r$的整数解（i, j）：

$i=(i_1,i_2,\cdots,i_m)$, $j=(j_1,j_2,\cdots,j_m)$;

该解满足下述特定情况下的附加条件:

（1）若S < T 且 S δ T 则 i⩽j

（2）若S = T 且 S δ S 则 i < j

（3）若S < T 且 T δ S 则 i > j

依赖关系方程

- 如果依赖方程（丢番图方程）有满足上述条件的整数解（i，j），那么

  （1）若 i < j , 则S δ’ T

  （2）若 i > j , 则T δ’ S

  （3）若 i = j , 且 S = T ， 则S δ’ T

其中δ’ 表示间接依赖关系。

对于多变量线性丢番图方程，即方程形式如下：

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

其中 $a_1, a_2, \ldots, a_n$ 和 b 是给定的整数，而 $x_1, x_2, \ldots, x_n$ 是需要找到的整数解。这样的方程有解的充要条件是 b 能够被 $a_1, a_2, \ldots, a_n$ 的最大公约数整除。

依赖关系方程

考虑如下程序段:

```
L1 :   for I = 1 to 50 do
              . . .
    S :    X(2*I) = . . .
              . . .
    T :     . . . = . . . X(3*I + 1 ) . . .
              . . .
        endfor
```

这里：$f_1(I) = 2 * I$ ； $g_1(J) = 3 * J + 1$ 。

依赖方程为：$f_1(I) - g_1(J) = 0$ ➔ $2*I – 3*J = 1$,

而依赖约束为：$1 \leqslant I \leqslant 50$ ， $1 \leqslant J \leqslant 50$。

该方程的解（I,J）对应的数组变量会导致S和T之间的依赖。

数据依赖与自动并行

- 并行编译器
- 数据依赖关系
- **循环的向量化与并行化**

循环向量化

- 循环向量化

  将仅含有数组赋值语句的循环L转换成等价的向量语句

如：循环

```
    for I = 1 to N do
S:    A(I) = D(I) * E
T:    C(I) = A(I) + B(I)
    endfor
```

可以改写为**等价**的向量语句：

```
S': A(1:N) = D(1:N) * E
T': C(1:N) = A(1:N) + B(1:N)
```

循环向量化

- 可向量化循环

  如果将循环内的数组赋值改为相应的向量语句后，按原来语句次序执行所得结果与原来串行执行一样，那么...

但以下循环不可向量化：

```
    for I = 1 to N do
S:      A(I) = A(I-1) + 1; //不能写成A(1:N) = A(0:N-1) + 1
    endfor
```

而以下循环却可以向量化：

```
    for I = 1 to N do
S1:     A(I) = A(I+1) + 1; //可以写成A(1:N) = A(2:N + 1) + 1
    endfor
```

为什么？

循环向量化

- 可向量化循环的充要条件

  对于循环L=(L_1,L_2,. . ., L_m)其最内层循环L_m可向量化当且仅当：L_m中任意两个语句S和T，

  （1） 当S<T时，不存在方向向量为(0,0,…,1)的S对T的依赖关系，即不存在T δ S； （三种依赖关系中任何一种）

  （2） 当S=T时，不存在方向向量为(0,0,…,1)的S对T的**流依赖关系**，即不存在T δ^f S；

循环向量化

- 可向量化循环的充要条件

  换言之，最内层循环中如果不存在与语句顺序相反的依赖关系（即反向依赖），则最内层循环可向量化。

再次考察：

```
    for I = 1 to N do
S:      A(I) = A(I-1) + 1; //不能写成A(1:N) = A(0:N-1) + 1
    endfor
```

此循环中存在 S δ^f S，且方向为(1),距离为(1)

循环向量化示例（1）

考查以下循环可向量化的情况。

```
（1）for I = 2 to N – 1 do
          for J = 2 to N – 1 do
 S :          A(I, J) = B(I-1, J) + C
 T :          B(I, J) = A(I, J+1) * 2
          endfor
      endfor
```

如果向量化内层的J循环，则形成：

```
     for I = 2 to N – 1 do
 S:      A(I, 2:N-1) = B(I-1,2:N-1) + C
 T:      B(I, 2:N-1) = A(I, 3:N) * 2
     endfor
```

此向量化运算的结果不对！

- 存在依赖T δ^f S，方向为（1, 0）
- 存在依赖T δ^a S，方向为（0, 1）

循环向量化示例（1）

考查以下循环可向量化的情况。

```
（1）for I = 2 to N – 1 do
          for J = 2 to N – 1 do
S :           A(I, J) = B(I-1, J) + C
T :           B(I, J) = A(I, J+1) * 2
          endfor
      endfor
```

如果向量化内层的J循环，则形成：

```
    for I = 2 to N – 1 do
S:      A(I, 2:N-1) = B(I-1,2:N-1) + C
T:      B(I, 2:N-1) = A(I, 3:N) * 2
    endfor
```

此向量化运算的结果不对！

举例来说，在（I = 2，J = 2）的迭代中，原来的内层循环中语句T先行读取了A(2,3)的旧值来计算B(2,2)；而在向量化后的循环中，向量化的语句S则在（I = 2）迭代中先行更新了A(2,3)的值；而随后向量化语句T则只能使用新的A(2,3)来计算，是错误的。

循环向量化示例（2）

考查以下循环可向量化的情况。

（2） for I = 1 to N do
　　for J = 1 to N do
S : D(I, J) = A(I, J) + C
T : A(I+1, J+1) = B(I, J) * 2
　　endfor
　endfor

存在依赖T δ^f S，
方向为（1，1）

尝试向量化内层循环如下：

　for I = 1 to N do
S : D(I, 1:N) = A(I, 1:N) + C
T : A(I+1, 2:N+1) = B(I, 1:N) * 2
endfor

向量化正确吗？
不存在（0，1）的T δ^f S

循环向量化示例（2）

(2) for I = 1 to N do
  for J = 1 to N do
S : D(I, J) = A(I, J) + C
T : A(I+1, J+1) = B(I, J) * 2
  endfor
endfor

部分展开循环如下：

I=1时：J = 1,2,…N

S(1,1): D(1,1) = A(1,1) …

T(1,1): A(2,2) = B(1,1)…

S(1,2): D(1,2) = A(1,2)…

T(1,2): A(2,3) = B(1,2)…

向量化为：
A(2,2:N+1) = B(1,1:N)…

向量化为：
D(1,1:N) = A(1,1:N)…

…(外层 I 循环依然串行执行，保持I层依赖关系，如 A(2,2)的写与读。)

I=2时：J=1,2,…,N

S(2,1): D(2,1) = A(2,1) …

T(2,1): A(3,2) = B(2,1)…

S(2,2): D(2,2) = A(2,2)…

T(2,2): A(3,3) = B(2,2)…

向量化为：
A(3,2:N+1) = B(2,1:N)…

向量化为：
D(2,1:N) = A(2,1:N)…

…

循环并行化

- 循环并行化

  将循环的迭代空间划分成不同的子集，分布到不同的处理机上执行（此时对各迭代子集的执行次序不作要求）。一般用doall(或 par-do)表示将循环并行化。

- 可并行化循环

  如果一个循环的各个迭代（子集）可按任意次序执行而结果与串行执行相同的话。

  以下循环可以并行化:

```
for I = 1 to N do                    doall I = 1 to N
   A(I) = A(I) + B(I)        ⟹          A(I) = A(I) + B(I)
endfor                               enddoall
```

循环并行化

- 可并行化循环的充要条件

  循环L=(L_1,L_2,. . ., L_m)中循环L_k可并行化当且仅当循环L中不存在层次为k的依赖关系（由K层携带的依赖关系），

  即不存在方向向量为（0,…,0,1_k,*,…,*)(含k-1个前导零）的依赖关系。

循环并行化

- 考察以下循环:

```
for  I = 1 to N do
  for J = 1 to M do
    X(I, J) = X(I, J-1) + X(I, J+1)
  endfor
endfor
```

存在方向为(0,1)的依赖关系

```
L1:   for  I = 0 to 4 do
L2:       for J = 0 to 4 do
    S:      X(I+1, J+2) = Y(I, J) + 1
    T:      Y(I+2, J+1) = X(I, J) + 1
          endfor
       endfor
```

存在方向为(1,1)的依赖关系:
S δ T 和 T δ S

循环并行化

```
for  I = 1 to N do
  for J = 1 to M do
    X(I, J) = X( I, J-1) + X(I, J+1)
  endfor
endfor
```

并行化外层循环I

```
doall  I = 1 to N
  for J = 1 to M do
    X(I, J) = X( I, J-1) + X(I, J+1)
  endfor
enddoall
```

```
L1:  for  I = 0 to 4 do
L2:      for J = 0 to 4 do
   S:      X(I+1, J+2) = Y( I, J) + 1
   T:      Y(I+2, J+1) = X(I, J) + 1
         endfor
       endfor
```

并行化内层循环J

```
L1:  for  I = 0 to 4 do
L2:      doall J = 0 to 4
   S:      X(I+1, J+2) = Y( I, J) + 1
   T:      Y(I+2, J+1) = X(I, J) + 1
         enddoall
       endfor
```

循环变换

- 循环变换的主要目的在于发掘程序中更多的并行性
- 循环变换是改变循环中语句实例的执行次序但不改变语句实例集合的程序变换技术。
- 对于循环L中的语句实例S(i)和T(j)，如果存在T(j)依赖于S(i)，且在变换后的循环L'中仍然有T(j)的执行迟于S(i)，那么称该循环变换合法，循环L和L'等价。
- 循环变换技术很多，如　循环交换、语句重排、循环分布与置换等等。

循环分布

- 循环分布（loop distribution）

  语句级变换，将一个循环分解为多个小循环，每个小循环和原来的循环有相同的迭代空间，只是包含的语句少些。该变换主要用于：

  - 分解出可向量化或可并行化的循环
  - 将原循环分解为较小循环以改善cache局部性
  - 创建紧嵌套循环
  - 分解原循环为若干使用较少变量的循环以提高寄存器的使用效率

  该变换根据语句依赖图进行其上的强连通子图分类，得到所谓凝聚图，并按其中的偏序关系执行分解后的循环

循环分布

考虑如下循环：

L1： for I = 4 to 100 do

S1 : A(I) = B(I-2) + 1

S2: C(I) = B(I-1) +F(I)

S3: B(I) = A(I-1) + 2

S4: D(I) = D(I+1) + B(I-1)

endfor

依赖关系如下：

S1 δ^f S3

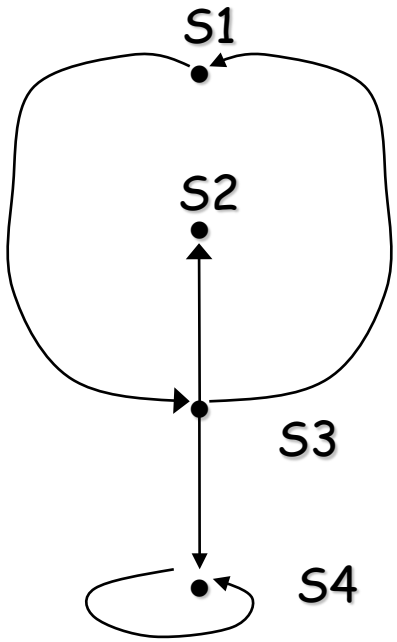
S3 δ^f S1

S3 δ^f S2

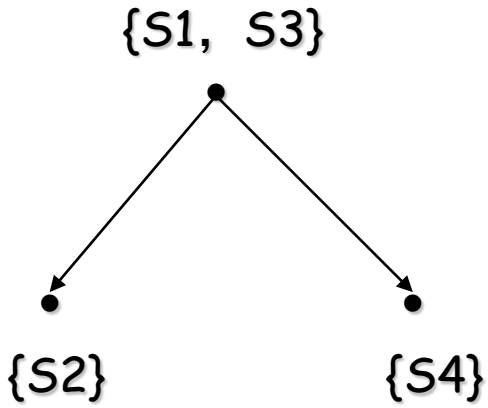
S3 δ^f S4

S4 δ^a S4

循环分布



语句依赖图



凝聚图

循环分布

- 这样把原循环分解成两大部分：

```
L1:   for I = 4 to 100 do
        S1 :   A(I) = B(I-2) + 1
        S3:    B(I) = A(I-1) + 2
      endfor
```

```
L2:   for I = 4 to 100 do
        S2:   C(I) = B(I-1) +F(I)
      endfor
L3:   for I = 4 to 100 do
        S4:  D(I) = D(I+1) + B(I-1)
      endfor
```

语句实例执行次序为：

（1）首先执行循环L1

（2）L1执行完后，可同时执行循环L2和L3

此外：

循环L2可以并行化执行；

循环L3可以向量化执行；

语句重排

- 语句重排（statement reordering)基于语句依赖图的程序变换，它改变语句的词法顺序但不改变语句间依赖关系。
- 该变换常用于循环向量化。当循环中语句间有循环依赖关系时，可通过此变换将与语句执行顺序相反的依赖关系改变为与语句执行顺序一致的依赖关系，从而使循环向量化。

语句重排

- 考虑如下循环例1:

  L : for I = 2 to N do

  S1 : A(I) = B(I) + C(I+1)

  S2: D(I) = A(I+1) + 1

  S3: C(I) = D(I)

  endfor

- 上述循环中依赖关系为:

  S2 δ^a S1, S1 δ^a S3, 和 S2 δ^f S3

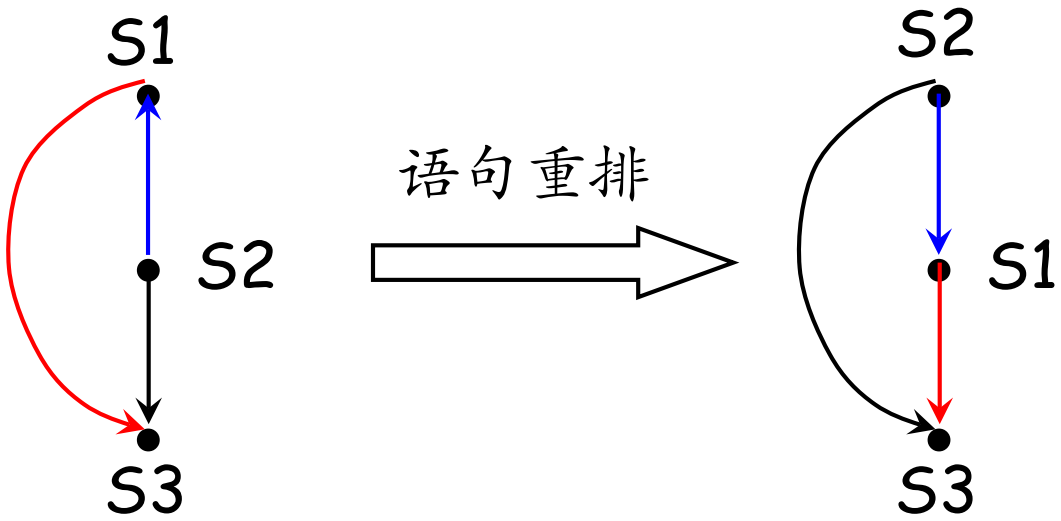
其中依赖关系S2 δ^a S1使得循环不能向量化。

语句重排

```
for I = 2 to N do
S1 :  A(I) = B(I) + C(I+1)
S2:   D(I) = A(I+1) + 1
S3:   C(I) = D(I)
endfor
```

重排+向量化

```
S2:   D(2:N) = A(3:N+1) + 1
S1 :  A(2:N) = B(2:N) + C(3:N+1)
S3:   C(2:N) = D(2:N)
```



原语句依赖图　　重排后的语句依赖图

循环置换

- 循环置换(loop permutation)是改变循环位置的程序变换，属于迭代级的变换，而循环交换即是此种变换的特例。该变换有如下特点：
  - 置换外层无循环依赖的循环与内层有依赖的循环，使得置换后内层可向量化；
  - 置换无依赖的循环到外层使得整个循环嵌套可以并行执行，可增加每次迭代的并行粒度并减少障碍同步次数；
  - 在有多层可向量化的循环时，置换范围较大的循环到外层可以增加向量的长度。

循环置换

- 考虑如下循环例2:

```
L1 : for I = 1 to M do
L2:     for J = 1 to M do
  S :       B(I,J) = A(I,J-1)
  T :       A(I,J) = B(I,J) * C(I,J)
          endfor
       endfor
```

循环中存在(0,1)的依赖关系TδS，不能对内层向量化。可以利用循环置换（交换）：

```
L2: for J = 1 to M do
L1:     for I = 1 to M do
```

此时，上述循环嵌套在内层可以向量化

循环置换

- 考虑以下循环例3：

```
    for I = 2 to N do
      for J = 2 to N do
S:        A(I,J) = ( A(I-1,J) + A(I+1,J) ) /2
      endfor
    endfor
```

此循环中依赖关系：方向向量为(1,0)的S δ^f S和(1,0)的S δ^a S;
因此循环嵌套中外层不可并行，
内层可以并行(无依赖关系），但并行粒度仅为一条语句。
可以采用交换循环的方式：

```
    for J = 2 to N do
      for I = 2 to N do
```

此时外层循环可以并行，其粒度为一个（内层）循环。

循环置换

- 循环置换的充要条件：

  假定P是mXm的置换矩阵，由P定义的m层循环嵌套L的置换是合法的，当且仅当对于L的每一个方向向量σ，均有 σP > 0 成立。

  mXm的置换矩阵P定义为：

  - 每个元素非0即1
  - 每行有且仅有一个元素为1
  - 每列有且仅有一个元素为1

  σP > 0的意思是向量第一个非0元素为正

  令π(i)表示P中第i列中为1的元素所在的行号，则函数π：i →π(i)是集合{1,2,…,m}上的一个置换，它完全确定矩阵P。P可以表示为：

$$P = \begin{pmatrix} 1, & 2, & \cdots, & m \\ \pi(1), & \pi(2), & \cdots, & \pi(m) \end{pmatrix} \text{ 或 } \begin{pmatrix} \pi(1), \pi(2), \cdots, \pi(m) \end{pmatrix}$$

循环置换

- 考虑循环置换例1和例2:

  对于例2，置换矩阵P = [2, 1] , 而原循环中的方向向量为σ = (0,1), σP = (0,1) [2,1] = (1,0) > 0。因此该循环交换是合法的。

  对于例3，置换矩阵P = [2, 1] , 而原循环中的方向向量为σ = (1,0), σP = (1,0) [2,1] = (0,1) > 0。因此该循环交换是合法的。

  这里P=[2,1] 其矩阵形式为:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

  而P'= [3, 2, 1]的矩阵形式为:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

循环逆转

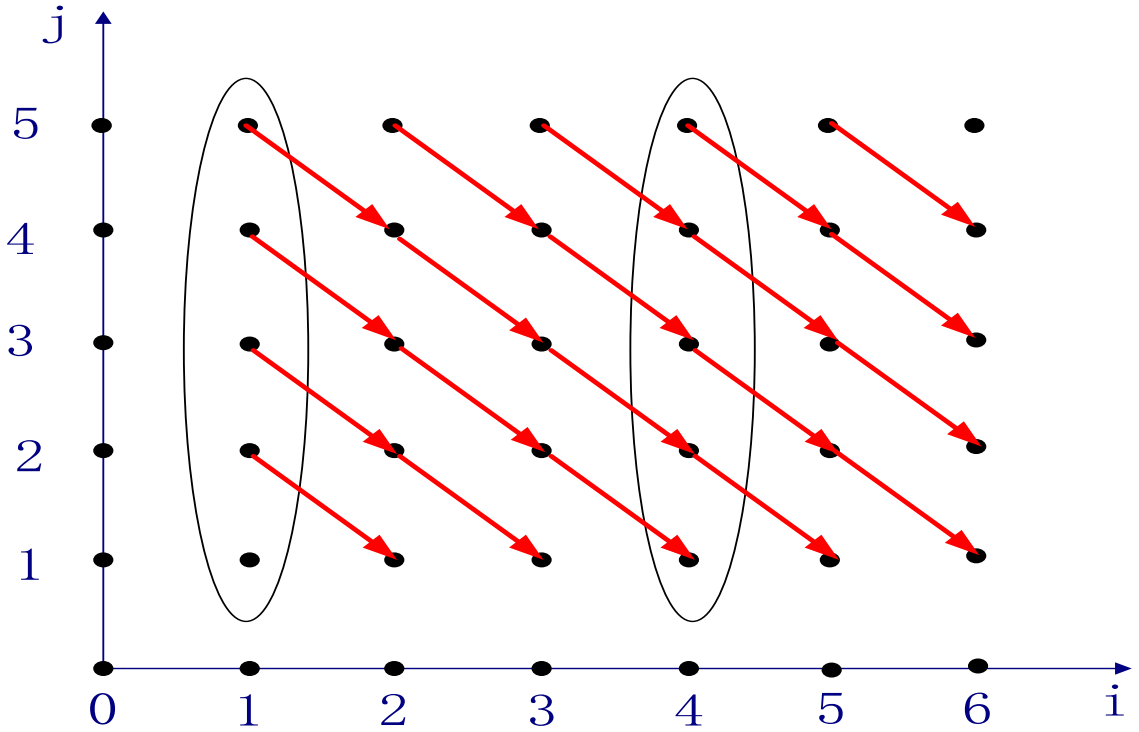
- 循环逆转(loop reversal) – **颠倒循环中迭代执行的顺序，改变了循环迭代方向的变换，也使得变换循环中方向向量发生逆转**。
- 如果循环在逆转变换后，它的方向向量均为正向量，则称该变换前后的循环等价，该变换是合法的。

考虑如下循环例4:

```
for I = 1 to 100 do
    for J = 1 to 5 do
S:      A(I,J) = A(I-1,J+1) +1
    endfor
endfor
```

循环逆转

- 例4迭代依赖图如下，可知它含有方向向量为(1,-1)的依赖关系，其内层循环可以并行化（但粒度为5次迭代），其外层不能并行化，也不能进行循环交换（为什么？）

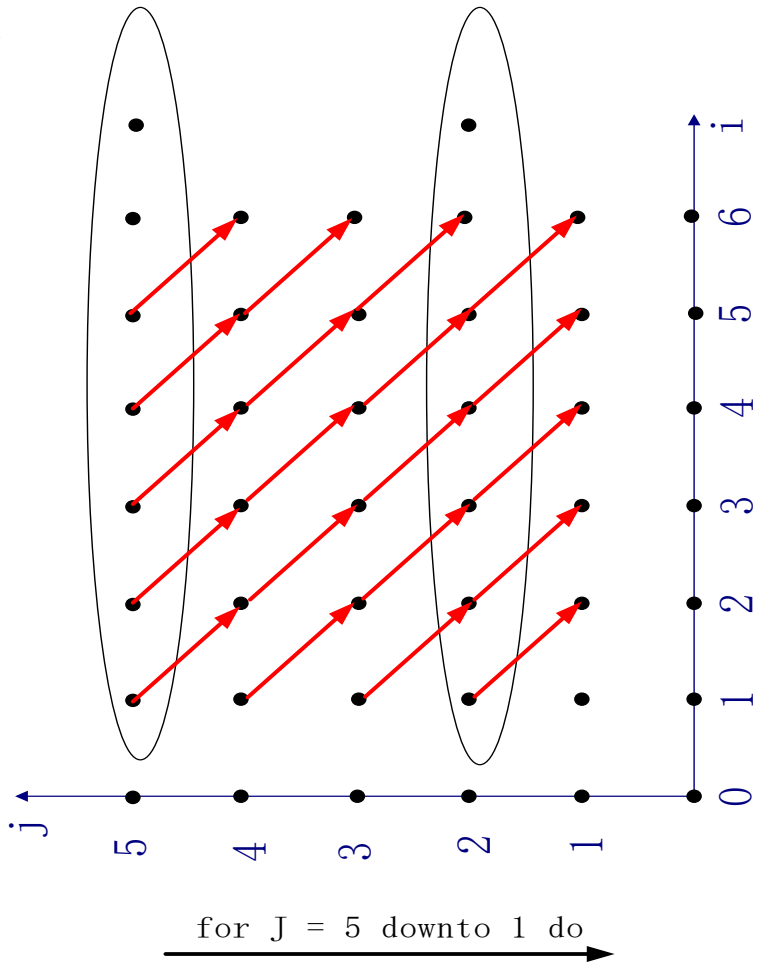


对循环J进行并行，粒度为5次迭代

循环逆转

- 对例4循环J进行逆转，则方向向量变为(1,1)。
- 可以对循环嵌套进行循环交换。此时内层循环I可以并行化（粒度为100次迭代）
- 迭代依赖图如右所示：

```
for J = 5 downto 1 do
      for I = 1 to 100 do
S:          A(I,J) = A(I-1,J+1) +1
      endfor
endfor
```

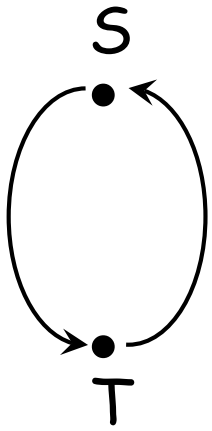


圈收缩

- 圈收缩(cycle shrinking) – 此变换技术一般用于依赖距离大于1的循环中，它将一个串行循环分成两个紧嵌套循环，其中外层依然串行执行，而内层则是并行执行（一般粒度较小）。

考虑如下循环例5：(K 为正整数)

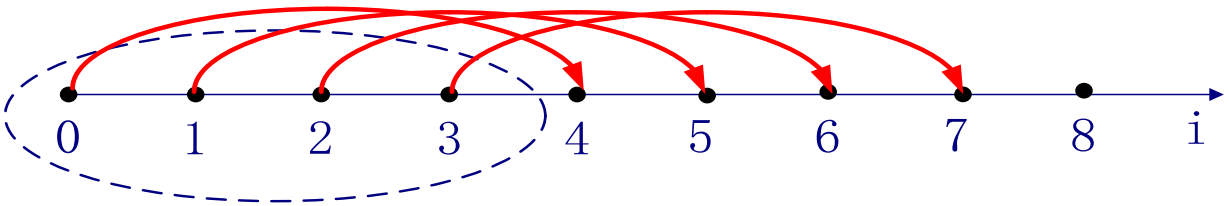
```
    for I = 0 to N do
S:      A(I+K) = B(I)
T:      B(I+K) = A(I) + C(I)
    endfor
```



语句依赖图

圈收缩

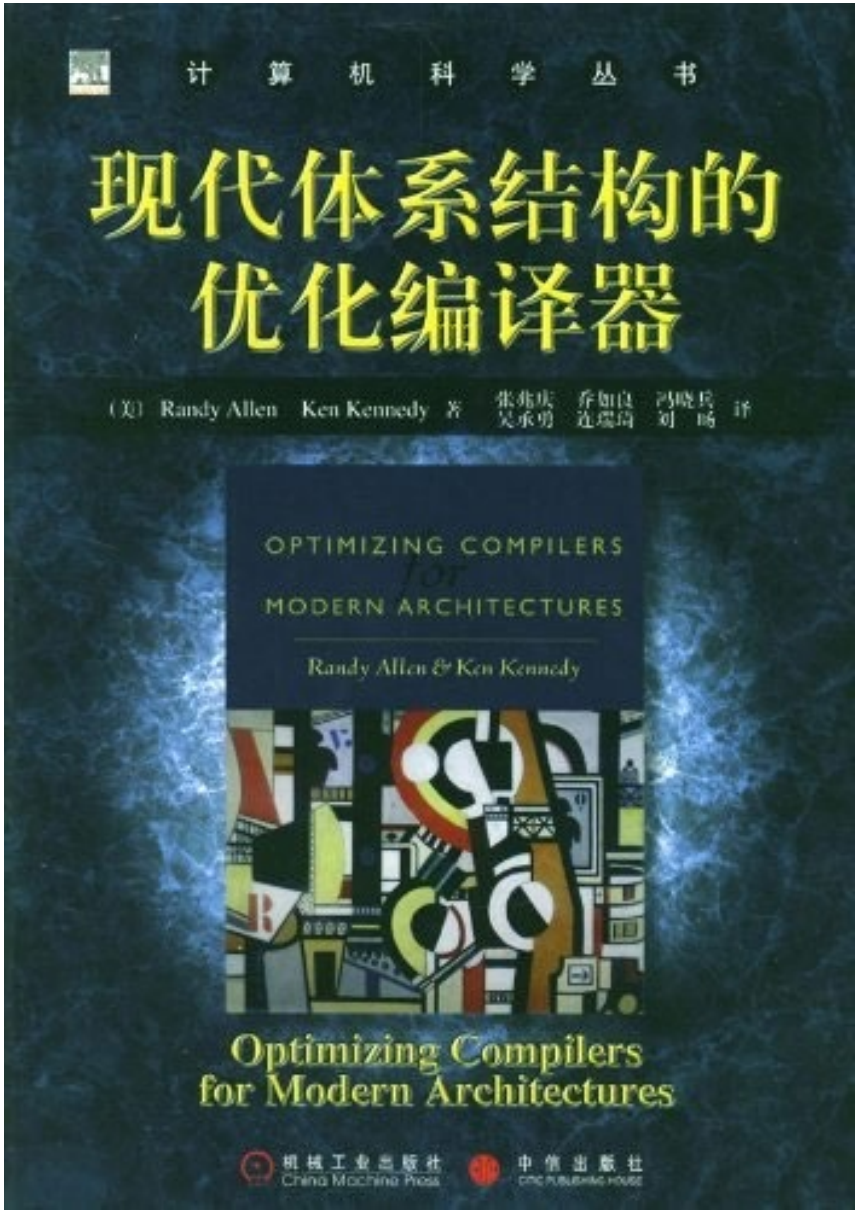
- 循环例5既不能向量化，也不能并行化（why？）
  考察（K=4时）迭代依赖图：



可知，此循环可以按K个迭代一组串行执行，而每一组内可以并行执行。循环例5可以改写为：

```
for J = 0 to N step K
        doall I = J to J+K-1
S:          A(I+K) = B(I)
T:          B(I+K) = A(I) + C(I)
        enddoall
endfor
```

参考资料


计 算 机 科 学 丛 书
现代体系结构的优化编译器
(美) Randy Allen Ken Kennedy 著
OPTIMIZING COMPILERS
for
MODERN ARCHITECTURES
Randy Allen & Ken Kennedy
Optimizing Compilers
for Modern Architectures
机械工业出版社
China Machine Press
中信出版社