CS 610: Dependence Testing

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2024-25-I

How to Write Efficient and Scalable Programs?

Good choice of algorithms and data structures

Determines the number of operations executed

Code that the compiler and architecture can effectively optimize

Determines the number of instructions executed

Proportion of parallelizable and concurrent code

Amdahl's law

Specialize to the target architecture platform

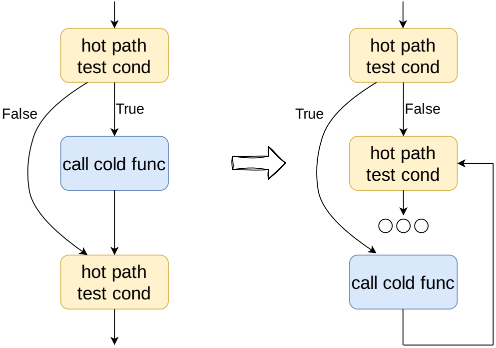
Memory hierarchy, cache sizes, advanced features like AMX

Role of a Good Parallelizing Compiler

Try and extract performance automatically

Optimize memory access latency

- Code restructuring optimizations (e.g., loop interchange)
- Prefetching optimizations (e.g., software prefetching)
- Data layout optimizations
- Code layout optimizations



Machine code layout optimizations

Parallelism Challenges for a Compiler

On single-core machines

Focus is on register allocation, instruction scheduling, reducing the cost of array accesses

On parallel machines

- Find **parallelism** in sequential code, find portions of work that can be executed in parallel
- Principle strategy is data decomposition—good idea because data parallelism can scale

Can we parallelize the following loops?

Focus is on loop parallelism because it can provide more savings

Inter-statement and intra-statement parallelism is limited

```
DO I = 1, 100
  A(I) = A(I) + 1
```

i	R	W
1	A(1)	A(1)
2	A(2)	A(2)
3	A(3)	A(3)

unroll

```
DO I = 1, 100
  A(I) = A(I-1) + 1
```

i	R	W
1	A(0)	A(1)
2	A(1)	A(2)
3	A(2)	A(3)

Data Dependences

```
S1    a = b + c
S2    d = a * 2
S3    a = c + 2
S4    e = d + c + 2
```

Execution constraints

- S2 must execute after S1
- S3 must execute after S2
- S3 must execute after S1
- S3 and S4 can execute concurrently (in any order)

There is a **data dependence** from S1 to S2 if and only if

(i) Both statements access the same memory location,

(ii) At least one of the accesses is a write,

(iii) There is a feasible execution path at run-time from S1 to S2.

Types of Dependences Based on Memory Accesses

Flow (a.k.a. true or RAW)
(denoted by $S_1 \delta S_2$)

```
S1    X = ...
S2    ... = X
```

Anti (a.k.a. WAR)
(denoted by $S_1 \delta^{-1} S_2$)

```
S1    ... = X
S2    X = ...
```

Output (a.k.a. WAW)
(denoted by $S_1 \delta^{o} S_2$)

```
S1    X = ...
S2    X = ...
```

Bernstein's Conditions

- Suppose there are two processes P_1 and P_2
- Let I_i be the set of all input variables for the process P_i
- Let O_i be the set of all output variables for the process P_i
- P_1 and P_2 can execute in parallel (denoted by $P_1 || P_2$) if and only if
  - $O_1 \cap I_2 = \phi$
  - $O_2 \cap I_1 = \phi$
  - $O_1 \cap I_2 = \phi$

Two processes can execute in parallel if they are flow-, anti-, and output-independent

- If $P_i || P_j$, does that imply $P_j || P_i$?
- If $P_i || P_j$ and $P_j || P_k$, does that imply $P_i || P_k$?

A. Bernstein. Analysis of Programs for Parallel Processing. IEEE Transactions on Electronic Computers, 1966.

Finding Parallelism in Loops—Is it Easy?

Need to check whether two array subscripts access the same memory location

```
      for i = 1 to N
S1     A[i+1] = A[i] + B[i]
```

```
      for i = 1 to N
S1     A[i+4] = A[i] + B[i]
```

- Statement S1 depends on itself in both examples, however, there is a subtle difference
- Compilers need formalism to analyze dependences and transform loops

Enumerate All Dependences in Loops

```
      for i = 1 to 50
S1      A[i] = B[i-1] + C[i]
S2      B[i] = A[i+2] + C[i]
```

Unrolling loops helps figure out dependences

S1(1)	A[1]	=	B[0]	+	C[1]
S2(1)	B[1]	=	A[3]	+	C[1]
S1(2)	A[2]	=	B[1]	+	C[2]
S2(2)	B[2]	=	A[4]	+	C[2]
S1(3)	A[3]	=	B[2]	+	C[3]
S2(3)	B[3]	=	A[5]	+	C[3]

- large loop bounds
- loop bounds may not be known at compile time

Normalized Iteration Number

Parameterize the statement with the loop iteration number

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


      DO I = L, U, S
S2      ...
```

For a loop where the loop index I runs from L to U in steps of S, the normalized iteration number of a specific iteration is $(I - L)/S + 1$, where I is the value of the index on that iteration

Iteration Vector and Lexicographic Ordering

Given a nest of n loops, the iteration vector i of an iteration of the innermost loop is a vector of integers containing the iteration numbers for each of the loops in order of nesting level.

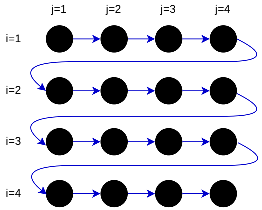
The iteration vector $\vec{i}$ is $\{i_1, i_2, \ldots, i_n\}$ where i_k, $1 \leq k \leq n$, represents the iteration number for the loop at nesting level k.

- A vector (d_1, d_2) is positive if $(0, 0) < (d_1, d_2)$, i.e., its first non-zero component is positive
- Iteration $\vec{i}$ precedes iteration $\vec{j}$, denoted by $\vec{i} < \vec{j}$, if and only if
  (i) $i[1:n-1] < j[1:n-1]$, or
  (ii) $i[1:n-1] = j[1:n-1]$ and $i_n < j_n$

Iteration Space Graphs

- Represents each dynamic instance of a loop as a point in the graph
- Arrows among points represent dependences

```
      for i = 1 to 4 do
        for j = 1 to 4 do
S1        A(i,j) = A(i,j-1) * x
```



Dimensions of an iteration space depends the loop nest depth, need not always be rectangular

```
      for i = 1 to 5 do
        for j = i to 5 do
S1        A(i,j) = B(i,j) + C(j)
```

Formal Definition of Loop Dependence

There is a loop dependence from S1 to S2 in a loop nest iff there exist two iteration vectors i and j such that

(i) $i < j$ or $i = j$ and there is a path from S1 to S2 in the body of the loop,

(ii) S1 accesses memory location M on iteration i and S2 accesses M on iteration j, and

(iii) One of these accesses is a write.

Distance and Direction Vectors

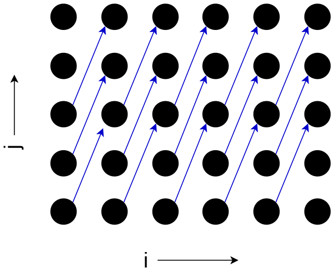
- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location
- Dependence distance vector $d(\vec{i}, j)$ is defined as a vector of length n such that $d(\vec{i}, j)_k = j_k - i_k$

```
DO i = 1, 6
  DO j = 1, 5
    A(i,j) = A(i-1,j-2) + 1
```

Distance vector = $(1, 2)$

outer loop

inner loop



Direction Vectors

Dependence direction vector $D(\vec{i}, j)$ is defined as a vector of length n such that

$$D(i,j)_k = \begin{cases} - \text{ if } D(i,j)_k < 0 \\ 0 \text{ if } D(i,j)_k = 0 \\ + \text{ if } D(i,j)_k > 0 \end{cases}$$

<	Positive
>	Negative
=	Zero
*	Mixed

alternate notation

Distance vector is a more **precise** form of a direction vector

For a valid dependence, the leftmost non-“0” component of the direction vector must be “+”

Summarizing Dependences

```
    DO J = 1, 10
     DO I = 1, 10
S1     A(I+1,J) = A(I,J) + 5
```

What are the dependences? How many?

The number of dependences between a pair of accesses is equal to the number of distinct direction vectors over all the dependences between those accesses

Distance and Direction Vector Examples

```
    DO I = 1, N
      DO J = 1, M
S1        A(I,J) = ...
S2        ... = A(I,J) + ...
```

```
    DO I = 1, N
      DO J = 1, M
S1        A(I,J) = A(I,1) + ...
```

```
    DO I = 1, N
      DO J = 1, M
        DO K = 1, L
S1        A(I+1,J,K-1) = A(I,J,K) + 10
```

```
  DO I = 1, N
    DO J = 1, M
S1    A(I,J) = A(I,J-3) + A(I-2,J) +
                A(I-1,J+2) + A(I+1,J-1)
```

Dependence Types

- There are two ways in which a statement S2 can depend on another statement S1, where both S1 and S2 are inside a loop
  - **Loop-carried**: S1 and S2 execute in different iterations
  - **Loop-independent**: S1 and S2 execute in the same iteration
- These types partition all possible data dependences

```
      DO I = 1, N
S1       A(I+1) = F(I)
S2       F(I+1) = A(I)
```

```
      DO I = 1, N
S1       A(I+1) = F(I)
S2       G(I+1) = A(I+1)
```

Loop-Carried and Loop-Independent Dependences

Loop-carried

- S1 references location M on iteration i
- S2 references M on iteration j
- $D(i,j) > 0$ (i.e., contains a "+" as leftmost non-"0" component)

```
    DO I = 1, 10
      DO J = 1, 10
        DO K = 1, 10
S1        A(I,J,K+1) = A(I,J,K)
```

Level of a loop-carried dependence is the leftmost non-"0" index of the dependence D(i,j) (denoted by S1δ_lS2)

Loop-independent

- S1 refers to location M on iteration i
- S2 refers to M on iteration j and $i = j$
- There is a control flow path from S1 to S2 within the iteration

```
    DO I = 1, 9
S1    A(I) = ...
S2    ... = A(10-I)
```

denoted by S1δ_∞S2

Having a common loop is not necessary

Program Transformations and Validity

Parallelism and Data Dependence

- Parallel loop iterations imply random interleaving of statements in the loop body
- Compilers apply transformations only when it is safe to do so

> A reordering transformation merely changes the order of execution of the code, without adding or deleting any executions of any statements

- A reordering transformation that preserves every dependence preserves the meaning of the program

Direction Vector Transformation

- Let T be a transformation applied to a loop nest
- Assume T does not rearrange the statements in the body of the loop
- T is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non-“0” component that is “-”

> A transformation is valid for the program to which it applies if it preserves all dependences in the program

Utility of Dependence Levels

- A reordering transformation preserves all level-k dependences if it
  (i) preserves the iteration order of the level-k loop,
  (ii) does not interchange any loop at level < k to a position inside the level-k loop, and
  (iii) does not interchange any loop at level > k to a position outside the level-k loop.

```
    DO I = 1, 10
S1    A(I+1) = F(I)
S2    F(I+1) = A(I)
```

```
    DO I = 1, 10
S2    F(I+1) = A(I)
S1    A(I+1) = F(I)
```

Statement order is irrelevant for loop-carried dependences but is important for loop-independent dependences

Are these transformations valid?

```
   DO I = 1, 10
     DO J = 1, 10
       DO K = 1, 10
S        A(I+1,J+2,K+3) = A(I,J,K) + B
```

```
   DO I = 1, 10
     DO K = 10, 1, -1
       DO J = 1, 10
S        A(I+1,J+2,K+3) = A(I,J,K) + B
```

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

```
     D(1) = A(1) + E
     DO I = 2, N
S1     A(I-1) = B(I-1) + C
S2     D(I) = A(I) + E
     A(N) = B(N) + C
```

Dependence Testing

Dependence Testing

Dependence testing is used to determine whether dependences exist between two subscripted references to the same array in a loop nest

Dependence question

Can $4 * I$ be equal to $2 * I + 2$ for $I \in [1, N]$?

```
DO I=1, N
  A(4*I) = ...
  ... = A(2*I+2)
```

affine

Given (i) two subscript functions f and g and (ii) lower and upper loop bounds L and U respectively, does $f(i_1) = g(i_2)$ have a solution such that $L \leq i_1, i_2 \leq U$?

Multiple Loop Indices, Multi-Dimensional Array

- Assumptions
  - Array subscripts are affine
  - Loops are in normalized form
- Let α and β be two valid vectors in the iteration space of the loop nest
- There is a dependence from S1 to S2 iff

$$\exists \alpha, \beta, \;\; \alpha \leq \beta \wedge f_i(\alpha) == g_i(\beta) \;\; \forall i, 1 \leq i \leq m$$

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

Solving the system of equations for arbitrary functions f and g is NP-complete

Approximate Dependence Testing

- The following system of equations with $2n$ variables and m equations is the most common

$$
\begin{aligned}
a_{11}i_1 + a_{12}i_2 + \cdots + a_{1n}i_n + c_1 &= b_{11}j_1 + b_{12}j_2 + \cdots + b_{1n}j_n + d_1 \\
a_{21}i_1 + a_{22}i_2 + \cdots + a_{2n}i_n + c_2 &= b_{21}j_1 + b_{22}j_2 + \cdots + b_{2n}j_n + d_2 \\
&\cdots \\
a_{m1}i_1 + a_{m2}i_2 + \cdots + a_{mn}i_n + c_m &= b_{m1}j_1 + b_{m2}j_2 + \cdots + b_{mn}j_n + d_m
\end{aligned}
$$

- Solve the system of the form $Ax = B$ for integer solutions
    - A is a $m \times 2n$ matrix and B is a vector of m elements
- Finding solutions to Diophantine equations is NP-complete

Dependence Testing with GCD

- Coefficients of the loop indices are integers in Diophantine equations
- The Diophantine equation $a_1 i_1 + a_2 i_2 + \cdots + a_n i_n = c$ has an integer solution iff $\gcd(a_1, a_2, \ldots, a_n)$ evenly divides c
  - If there is a solution, we can test if it lies within the loop bounds
  - If not, then there is no dependence

```
     for i = 1 to N
S1     a[x*i+k] = ...
S2     ... = a[y*i+m];
```

$x * i_1 + k = y * i_2 + m$,
where $0 \le i_1, i_2 \le N$

- If $\mathrm{GCD}(x, y)$ divides $(m - k)$, then a dependence may exist between S1 and S2
- If $\mathrm{GCD}(x, y)$ does not divide $(m - k)$, then S1 and S2 are independent and can be executed in parallel

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

Problems with Dependence Testing with GCD

- Coefficients of the loop indices are integers in Diophantine equations
- The Diophantine equation $a_1 i_1 + a_2 i_2 + \cdots + a_n i_n = c$ has an integer solution iff $\gcd(a_1, a_2, \ldots, a_n)$ evenly divides c
  - If there is a solution, we can test if it lies within the loop bounds
  - If not, then there is no dependence

```
     for i = 1 to 10
S1     a[i] = b[i]+c[i]
S2     d[i] = a[i-100];
```

Problems

- Provides no information on distance or direction of dependence, only tells if there are no dependences
- Ignores loop bounds and GCD is often 1, resulting in false dependences

Lamport Test

- Used when there is a single index variable in the subscripts and the coefficients of the index variables are the same
- There is an integer solution only if $d = \frac{c_1 - c_2}{b}$ is an integer
  - Dependence is valid if $|d| \leq U_i - L_i$

```
A[...,b*i+c₁,...] = ...
... = A[...,b*i+c₂,...]
```

```
    for i = 1 to n
      for j = 1 to n
S1      a[i,j] = a[i-1,j+1]
```

```
    for i = 1 to n
      for j = 1 to n
S1      a[i,2j] = a[i-1,2j+1]
```

Classifying Subscripts

- A subscript is a pair of subscript positions in a pair of array references
  - $A(i,j) = A(i,k) + C$
  - $\langle i, i \rangle$ is the first subscript, $\langle j, k \rangle$ is the second subscript
- A subscript is said to be
  - Zero index variable (ZIV) if it contains no index variable
  - Single index variable (SIV) if it contains only one index variable
  - Multi index variable (MIV) if it contains more than one index variable
- Consider $A(5, i+1, j) = A(1, i, k) + C$
  - First subscript is ZIV, second subscript is SIV, third subscript is MIV

Separability and Coupled Subscript Groups

- A subscript is **separable** if its indices do not occur in other subscripts
- If two different subscripts contain the same index they are **coupled**
  - $A(i+1, j) = A(k, j) + C$: Both subscripts are separable
  - $A(i, j, j) = A(i, j, k) + C$: Second and third subscripts are coupled

- Coupling indicates complexity in dependence testing

```
    DO I = 1, 100
S1    A(I+1,I) = B(I) + C
S2    D(I) = A(I,I) * E
```

Overview of Dependence Testing

(i) Partition subscripts of a pair of array references into separable and coupled groups
(ii) Classify each subscript as ZIV, SIV, or MIV
(iii) For each separable subscript apply single subscript test
   - If not done, go to next step
(iv) For each coupled group apply multiple subscript tests like Delta Test
(v) If still not done, merge all direction vectors computed in the previous steps into a single set of direction vectors

G. Goff et al. Practical Dependence Testing, PLDI'91.

A. Chauhan. Dependence Testing.

Simple Subscript Tests

- ZIV test
  - $e1$ and $e2$ are constants or loop invariant symbols
  - If $e1 \neq e2$, then no dependence exists
- SIV test
  - Strong SIV test: $\langle a * i + c_1, a * i + c_2 \rangle$
    - a,c_1,c_2 are constants or loop invariant symbols
    - Example: $\langle 4i + 1, 4i + 5 \rangle$
    - Solution: $d = (c_2 - c_1)/a$ is an integer and $|d| \leq |U_i - L_i|$
  - Weak SIV test: $\langle a_1 * i + c_1, a_2 * i + c_2 \rangle$
    - a_1,a_2,c_1,c_2 are constants or loop invariant symbols
    - Example: $\langle 4i + 1, 2i + 5 \rangle$ or $\langle i + 3, 2i \rangle$

```
DO j = 1,100
  A(e1) = A(e2) + B(j)
```

Weak SIV Test

- Weak zero SIV: $\langle a_1 * i + c_1, c_2 \rangle$
  - Solution: $i = (c_2 - c_1)/a_1$ is an integer and $|i| \leq |U - L|$

```
    DO I = 1, N
S1     Y(I,N) = Y(1,N) + Y(N,N)
```

```
    Y(1,N) = Y(1,N) + Y(N,N)
    DO I = 2, N-1
S1     Y(I,N) = Y(1,N) + Y(N,N)
    Y(N,N) = Y(1,N) + Y(N,N)
```

- Weak crossing SIV: $\langle a * i + c_1, -a * i + c_2 \rangle$
  - Solution: $i = (c_2 - c - 1)/2a$ is an integer and $|i| \leq |U - L|$

```
    DO I = 1, N
S1     A(I) = A(N-I+1) + C
```

```
    DO I = 1, (N+1)/2
S1     A(I) = A(N-I+1) + C
    DO I = (N+1)/2+1, N
S2     A(I) = A(N-I+1) + C
```

Other Dependence Tests

- Banerjee-Wolfe test: widely used test
- Power test: improves over Banerjee test
- Delta test: specializes in common array subscript patterns
- Omega test: “precise” test, most accurate for linear subscripts
- Range test: handles non-linear and symbolic subscripts

- Many variants of these tests exits

W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis, SC’91.

Banerjee-Wolfe Test

If the total subscript range accessed by *ref*1 does not overlap with the range accessed by *ref*2, then *ref*1 and *ref*2 are independent

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

[1 : 100]
[201 : 300]

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

[1 : 100]
[6 : 105]

```
for (k=0; k < N; k++) {
  c[f(i)] = ...;
  ... = c[g(j)];
}
```

True: $\exists i, j \in [0, N-1],\ i \leq j \wedge f(i) = g(i)$
Anti: $\exists i, j \in [0, N-1],\ i > j \wedge f(i) = g(i)$

```
for (k=0; k < N; k++) {
  ... = c[g(j)];
  c[f(i)] = ...;
}
```

True: $\exists i, j \in [0, N-1],\ i < j \wedge f(i) = g(i)$

Banerjee test

Delta Test

- Notation represents index values at the source and sink

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

- Let source iteration be denoted by I_0, and sink iteration be denoted by $I_0 + \Delta I$
- Valid dependence implies $I_0 + 1 = I_0 + \Delta I$
- We get $\Delta I = 1 \implies$ Loop-carried dependence with distance vector (1) and direction vector (+)

G. Goff et al. Practical Dependence Testing, PLDI'91.

Delta Test

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

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

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

- If a loop index does not appear in a subscript, its distance is unconstrained and its direction is "*" (denotes union of all 3 directions)
- Direction vector is (+, *)
  - (*, +) denotes (+, +), (0, +), (-, +)
  - (-, +) denotes a level 1 anti-dependence with direction vector (+,-)

Delta Test

Extract constraints from SIV subscripts and use them for other subscripts

```
DO I = 1, N
  A(I,I) = A(1,I-1) + C
```

```
DO I = 1, 100
  DO J = 1, 100
    A(I+1, I+J) = A(I, I+J-1) + C
```

```
DO I = 1, N
  A(I+1,I+2) = A(I,1) + C
```

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      A(J-I,I+1,J+K) = A(J-I,I,J+K)
```

Solving Integer Inequalities

- The loop nest inequalities specify a convex polyhedron
  - A polyhedron is convex if for two points in the polyhedron, all points on the line between them are also in the polyhedron
- Data dependence implies a search for integer solutions that satisfy a set of linear inequalities
  - Integer linear programming is an NP-complete problem
- Steps
  1. Use GCD test to check if integer solutions may exist
  2. Use simple heuristics to handle typical inequalities
  3. Use a linear integer programming solver that uses a branch-and-bound approach based on Fourier-Motzkin elimination for unsolved inequalities

Fourier-Motzkin Elimination

Input An n-dimensional polyhedron S with variables $x_1, x_2, \ldots x_n$

Goal Eliminate x_m, $m \leq n$

Output A polyhedron S' with variables $x_1, x_2, \ldots x_{m-1}, x_{m+1}, \ldots x_n$

Steps Let C be all constraints in S involving x_m

1. For every pair of a lower bound and upper bound on $x_m \in C$, such as, $L \leq c_1 x_m$ and $c_2 x_m \leq U$, create a new constant $c_2 L \leq c_1 U$
2. If integers c_1 and c_2 have a common factor, divide both sides by that factor
3. If the new constraint is not satisfiable, then there is no solution to S, i.e., S and S' are empty spaces
4. S' is the set of constraints $S - C$, plus the new constraints generated in Step 2

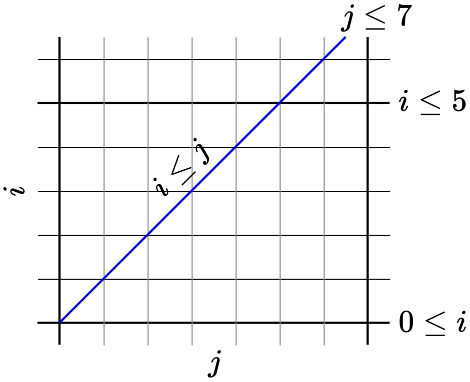
Example of Fourier-Motzkin Elimination

Consider the code

```
for (i = 0; i <= 5; i++)
  for (j = i; j <= 7; j++)
    Z[j,i] = 0;
```

Goal is to interchange the loops

```
for (j = __; j <= __; j++)
  for (i = __; i <= __; i++)
    Z[j,i] = 0;
```



Example of Fourier-Motzkin Elimination

```
for (i = 0; i <= 5; i++)
  for (j = i; j <= 7; j++)
    Z[j,i] = 0;
```

Use Fourier-Motzkin elimination to project the 2D space away from the i dimension and onto the j dimension

$$0 \le i \wedge i \le 5 \wedge i \le j \implies 0 \le j \wedge 0 \le 5,$$

and we already have $j \le 7$

The new constraints are:
$0 \le i, i \le 5, i \le j, 0 \le j, j \le 7$

Find the loop bounds from the original loop nest: $L_i : 0; U_i : 5, j; L_j : 0; U_j : 7$

```
for (j = 0; j <= 7; j++)
  for (i = 0; i <= min(5,j); i++)
    Z[j,i] = 0;
```

Use ILP for Dependence Testing

- **Algorithm**

  Input: A convex polyhedron S over variables $v_1, v_2, \ldots v_n$

  Output: "Yes" if S has an integer solution, "no" otherwise

```
for (i=1; i < 10; i++)
  Z[i] = Z[i+10];
```

Show that there are no two dynamic accesses i and i' with $1 \leq i \leq 9$, $1 \leq i' \leq 9$, and $i = i' + 10$.

Dependence Testing is Hard

- Most dependence tests assume affine array subscripts
- Unknown loop bounds can lead to false dependences
- Need to be conservative about aliasing
- Triangular loops adds new constraints
- Loop transformations can add additional variables

```
for (i=0; i < N; i++) {
  a[i] = a[i+10];
}
```

How do we compare *N* and 10?

```
for (i=0; i < N; i++) {
  for (j=0; j < i-1; j++) {
    a[i][j] = a[j][i];
}
```

Add $j < i$ as a new constraint

```
for (i=L; i < H; i++) {
  a[i] = a[i-1];
}
```

Loop transformations (e.g., normalization) add new variables

Why is Dependence Analysis Important?

- Dependence information is used to drive important loop transformations
- Goal is to remove dependences or parallelize in the presence of dependences
- We will discuss many transformations (e.g., loop interchange and loop fusion) next

References

- R. Allen and K. Kennedy. Optimizing Compilers for Multicore Architectures. Sections 1.1–1.6, Morgan Kaufmann.
- A. Aho et al. Compilers: Principles, Techniques, and Tools. Section 11.6, 2nd edition, Pearson Education.
- Qing Yi. Dependence Testing: Solving System of Diophantine Equations.