CS 610: Data Dependence Analysis

Swarnendu Biswas

Semester 2020-2021-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Copyright Information

How to Write Efficient and Scalable Programs?

Good choice of algorithms and data structures

- Determines number of operations executed

Code that the compiler and architecture can effectively optimize

- Determines number of instructions executed

Proportion of parallelizable and concurrent code

- Amdahl's law

Sensitive to the architecture platform

- Efficiency and characteristics of the platform
- For e.g., memory hierarchy, cache sizes

Role of a Good Compiler

Try and extract performance automatically

Optimize memory access latency

- Code restructuring optimizations
- Prefetching optimizations
- Data layout optimizations
- Code layout optimizations

Parallelism Challenges for a Compiler

- On single-core machines
  - Focus is on register allocation, instruction scheduling, reduce 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 since this can scale

Can we parallelize the following loops?

```
do i = 1, 100
  A(i) = A(i) + 1
enddo
```

```
do i = 1, 100
  A(i) = A(i-1) + 1
enddo
```

Dependences

S1 a = b + c

S2 d = a * 2

S3 a = c + 2

S4 e = d + c + 2

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 in any order, and concurrently

Data Dependence

- There is a data dependence from S1 to S2 if and only if
  - Both statements access the same memory location
  - At least one of the accesses is a write
  - There is a feasible execution path at run time from S1 to S2

Types of Dependences

Flow (true)	S1 X = … S2 … = X	S1 δ S2
Anti	S1 … = X S2 X = …	S1 δ^{-1} S2
Output	S1 X = … S2 X = …	S1 δ^{o} S2
Input	S1 … = a/b S2 … = b * c	

Bernstein's Conditions

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

Bernstein's Conditions

- Suppose there are two processes P_1 an
- Let I
  varia
- Let O_i be the set of all output variables for process P_i

- P_1 and P_2 can execute in parallel
  only
  - $O_2 \cap O_1 = \Phi$

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

Bernstein's Conditions

- Suppose there are two processes P_1 ar…
- Let I…
  varia…
- Let O_i be the set of all output variables for process P_i

- P_1 and P_2 can execute in parallel … only
  - $O_2 \cap O_1 = \Phi$

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 $P_i \,||\, P_k$?

Find Parallelism in Loops – Is it Easy?

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

Dependence in Loops

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

- Unrolling loops can help 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]
```

……

Dependence in Loops

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

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

- Unrolling loops can help 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]
```

Dependence in Loops

- Parameterize the statement with the loop iteration number

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

     DO I = L, U, S
S1     ...
     ENDDO
```

Normalized Iteration Number

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

Iteration Vector

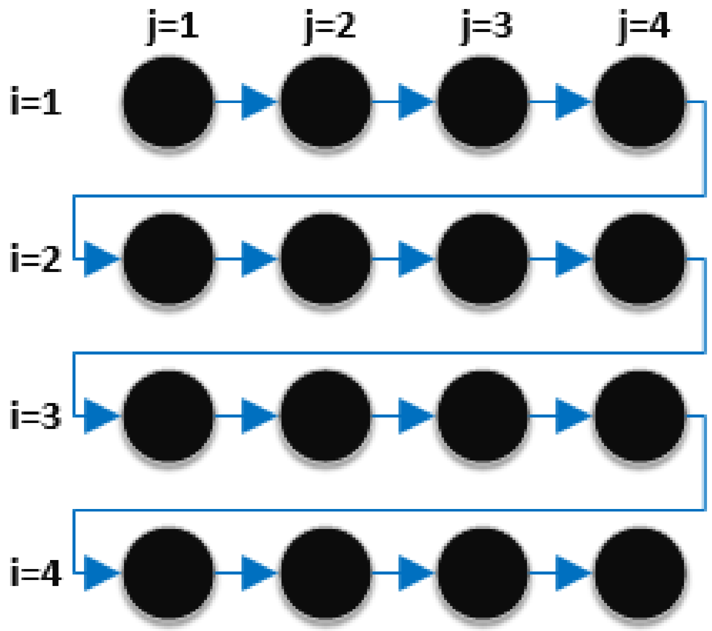
Given a nest of n loops, the *iteration vector* ***i*** of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.

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

Iteration Space Graphs

- Represent each dynamic instance of a loop as a point in the graph
- Draw arrows from one point to another to represent dependences

```
     for (i = 1; i <= 4; i++)
       for (j = 1; j <= 4; j++)
S1:      a[i][j] = a[i][j-1] * x;
```



Iteration Space Graph

- Dimension of iteration space is the loop nest level
- Not restricted to be rectangular

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

Lexicographic Ordering of Iteration Vectors

- Assume i is a vector, i_k is the k^{th} element of the vector i, and i[1:k] is a k-vector consisting of the leftmost k elements of i

- Iteration i precedes iteration j, denoted by i < 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 $i_n < j_n$

Formal Definition of Loop Dependence

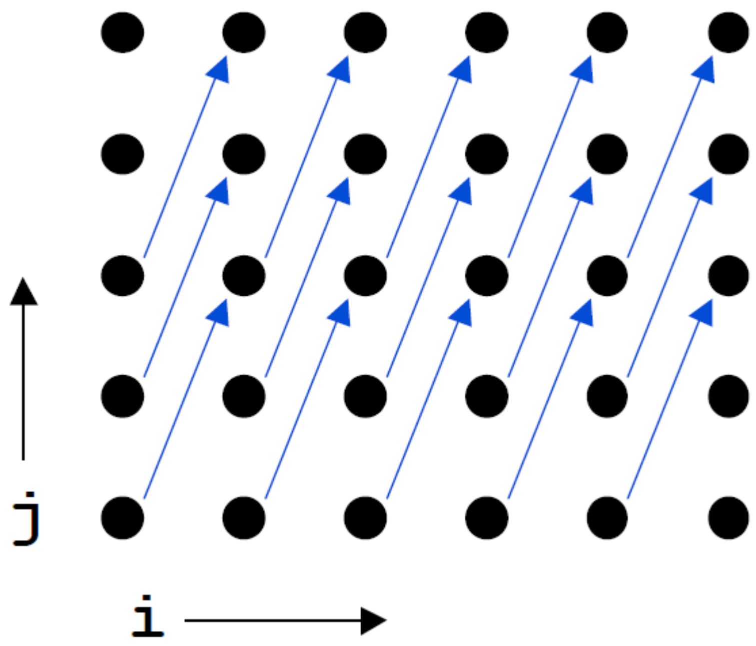
There exists a dependence from statement S1 to statement S2 in a common nest of loops if and only if there exist two iteration vectors i and j for the nest, such that

i. i < j or i = j and there is a path from S1 to S2 in the body of the loop,
ii. statement S1 accesses memory location *M* on iteration i and statement S2 accesses location *M* on iteration j, and
iii. one of these accesses is a write.

Distance Vectors

- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

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



outer loop

- Distance vector: (1, 2)

inner loop

Distance Vectors

- Suppose that there is a dependence from statement S1 on iteration i of a loop nest and statement S2 on iteration j, then the *dependence distance vector* d(i,j) is defined as a vector of length *n* such that $\boldsymbol{d}(\boldsymbol{i},\boldsymbol{j})_k = \boldsymbol{j}_k - \boldsymbol{i}_k$.

- A vector (d1, d2) is positive if (0,0) < (d1, d2), i.e., its first (leading) non-zero component is positive

Direction Vectors

- Suppose that there is a dependence from statement S1 on iteration i of a loop nest of *n* loops and statement S2 on iteration j, then the *dependence direction vector* is *D*(*i,j*) is defined as a vector of length *n* such that

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

Distance and Direction Vectors

- Suppose that there is a dependence from statement S1 on iteration i of a loop nest of *n* loops and statement S2 on iteration j, then the *depe*… *n* such…

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

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

Distance and Direction Vector Example

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

Distance and Direction Vector Example

```
    FOR I = 1, 5
      DO J = 1, 5
S1      A(I,J) = A(I,J-3) + A(I-2,J) + A(I-1,J+2) + A(I+1,J-1)
      ENDFOR
    ENDFOR
```

Program Transformations and Validity

Reordering Transformations

- A reordering transformation does not add or remove statements from a loop nest
  - Only reorders the execution of the statements that are already in the loop

Does not add or remove statements → Does not add or remove any new dependences

Validity of Dependence-Based Transformations

- A reordering transformation is said to be valid for the program to which it applies if it preserves all dependences in the program

Direction Vector Transformation

- Let *T* be a transformation applied to a loop nest
  - 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 “-”

Dependence Types

- If in a loop, statement S_2 depends on S_1, then there are two possible ways of this dependence occurring
  - S_1 and S_2 execute on different iterations - this is called a **loop-carried** dependence
  - S_1 and S_2 execute on the same iteration - this is called a **loop-independent** dependence
- These types partition all possible data dependences

Loop-Carried Dependences

- S1 can reference the common location on one iteration of a loop; on a subsequent iteration *S2* can reference the same location

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

i. S1 references location *M* on iteration i
ii. S2 references *M* on iteration j
iii. d(i,j) > 0 (that is, contains a “+” as leftmost non-“0” component)

Level of Loop-Carried Dependence

- The *level* of a loop-carried dependence is the index of the leftmost non-"0" of $D(i,j)$ for the dependence.

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

Utility of Dependence Levels

- A reordering transformation preserves all level-*k* dependences if it
  1. preserves the iteration order of the level-*k* loop
  2. does not interchange any loop at level < *k* to a position inside the level-*k* loop and
  3. does not interchange any loop at level > *k* to a position outside the level-*k* loop.

Utility of Dependence Levels

- A reordering transformation preserves all level-*k* dependences if it
  1. preserves the iteration order of the level-*k* loop
  2. does not interchange any loop at level < *k* to a position inside the level-*k* loop and
  3. 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)
    ENDDO
```

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

Is this transformation 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
      ENDDO
    ENDDO
  ENDDO
```

```
  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
      ENDDO
    ENDDO
  ENDDO
```

Loop-Independent Dependences

- S1 and S2 can both reference the common location on the same loop iteration, but with S1 preceding S2 during execution of the loop iteration.

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

i. S1 refers to memory location *M* on iteration i
ii. S2 refers to *M* on iteration j and i = j
iii. There is a control flow path from S1 to S2 within the iteration.

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

Validity of Transformations for Loop-Independent Dependences

- If there is a loop-independent dependence from S1 to S2, any reordering transformation that does not move statement instances between iterations and preserves the relative order of S1 and S2 in the loop body preserves that dependence.

Is this transformation valid?

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

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

Dependency Testing

Dependence Testing

- Dependence question
  - Can `4*I` be equal to `2*I+1` for `I` in `[1,N]`?

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

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 Nests

```
DO i=1,n
  DO j=1,m
    X(a₁*i + b₁*j + c₁) = …
    … = X(a₂*i + b₂*j + c₂)
  ENDDO
ENDDO
```

- Dependence test

$$a_1 * i_1 + b_1 * j_1 + c_1 = a_2 * i_2 + b_2 * j_2 + c_2$$
$$1 \leq i_1, i_2 \leq n$$
$$1 \leq j_1, j_2 \leq m$$

Multiple Loop Indices, Multi-Dimensional Array

```
DO i=1,n
  DO j=1,m
    X(a₁*i₁ + b₁*j₁ + c₁, d₁*i₁ + e₁*j₁ + f₁) = …
    … = X(a₂*i₂ + b₂*j₂ + c₂, d₂*i₂ + e₂*j₂ + f₂)
  ENDDO
ENDDO
```

- Dependence test

$$a_1 i_1 + b_1 j_1 + c_1 = a_2 i_2 + b_2 j_2 + c_2$$
$$d_1 i_1 + e_1 j_1 + f_1 = d_2 i_2 + e_2 j_2 + f_2$$
$$1 \leq i_1, i_2 \leq n$$
$$1 \leq j_1, j_2 \leq m$$

Multiple Loop Indices, Multi-Dimensional Array

```
DO i=1,n
  DO j=1,m
    X(a1*i1 + b1*j1 + c1, d1*i1 + e1*j1 + f1) = …
    … = X(a2*i2 + b2*j2 + c2, d2*i2 + e2*j2 + f2)
  ENDDO
ENDDO
```

- Dependence test

$$a_1 i_1 + b_1 j_1 + c_1 = a_2 i_2 + b_2 j_2 + c_2$$
$$d_1 i_1 + e_1 j_1 + f_1 = d_2 i_2 + e_2 j_2 + f_2$$
$$1 \le i_1, i_2 \le n$$
$$1 \le j_1, j_2 \le m$$

complex

$ax + by = c$

Data Dependence Testing

- Variables in loop indices are integers → Diophantine equations
- The Diophantine equation $a_1 i_1 + a_2 i_2 + \cdots + a_n i_n = c$ has an integer solution if and only if $\gcd(a_1, a_2, \ldots, a_n)$ 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

Complexity in Dependence Testing

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

Separability and Couple 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 can cause imprecision in dependence testing

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

Overview of Dependency Testing

1. Partition subscripts of a pair of array references into separable and coupled groups
2. Classify each subscript as ZIV, SIV or MIV
3. For each separable subscript apply single subscript test
    - If not done, goto next step
4. For each coupled group apply multiple subscript test like Delta Test
5. If still not done, merge all direction vectors computed in the previous steps into a single set of direction vectors

Simple Dependence Testing: Delta Notation

- Notation represents index values at the source and sink

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

- 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 $\Delta I = 1 \Rightarrow$ Loop-carried dependence with distance vector (1) and direction vector (+)

Simple Dependence Testing: Delta Notation

```
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: (+,0,-)

Simple Dependence Testing: Delta Notation

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

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

flow W → R

$I_0 + 1 = I_0 + \Delta I$

$\Rightarrow \Delta I = 1$

$(+, *) \rightarrow \delta$

1, 0

1, 1

(1, 2)

- The direction vector for the dependence is (+, *)

Simple Dependence Testing: Delta Notation

- * denotes union of all 3 directions

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

- (*, +) denotes { (+, +), (0, +), (-, +) }
  - (-, +) denotes a level 1 anti-dependence with direction vector (+, -)

Other Dependence Tests

- GCD test is simple but not accurate
  - It can tell us that there is no solution
- Other tests
  - Banerjee-Wolfe test: widely used test
  - Power Test: improvement over Banerjee test
  - Omega test: “precise” test, most accurate for linear subscripts
  - Range test: handles non-linear and symbolic subscripts
  - many variants of these tests

$2x + 2y = 4$

$\Rightarrow x + y = 2$

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

x[a*i+b]

x[c*i+d]

$\boxed{GCD(c,a)}$ (d-b)

x[2*i+3] = x[2*i]+5;

-3 | gcd(2,2)

Dependence Testing is Hard

- Unknown loop bounds can lead to false dependences
- Need to be conservative about aliasing
- Triangular loops add new constraints

Why is Dependence Analysis Important?

- Dependence information can be used to drive other important loop transformations
  - For example, loop parallelization, loop interchange, loop fusion

- We will see many examples soon

References

- R. Allen and K. Kennedy – Optimizing Compilers for Modern Architectures, Chapter 2.
- Michelle Strout – CS 553: Compiler Construction, Fall 2007.