Lecture 19
Array Dependence Analysis & Parallelization

[ALSU 11.6]

Data Dependence

\[
\begin{align*}
S_1 &: \quad A = 1.0 \\
S_2 &: \quad B = A + 2.0 \\
S_3 &: \quad A = C + D \\
\vdots \\
S_n &: \quad A = B/C
\end{align*}
\]

We define four types of data dependence.

- **Flow (true) dependence**: a statement \( S_i \) precedes a statement \( S_j \) in execution and \( S_i \) computes a data value that \( S_j \) uses.
- It implies that \( S_i \) must execute before \( S_j \).

We define four types of data dependence.

- **Anti dependence**: a statement \( S_i \) precedes a statement \( S_j \) in execution and \( S_i \) uses a data value that \( S_j \) computes.
- It implies that \( S_i \) must be executed before \( S_j \).

We define four types of data dependence.

- **Output dependence**: a statement \( S_i \) precedes a statement \( S_j \) in execution and \( S_i \) computes a data value that \( S_j \) also computes.
- It implies that \( S_i \) must be executed before \( S_j \).

We define four types of data dependence.

- **Input dependence**: a statement \( S_i \) precedes a statement \( S_j \) in execution and \( S_j \) computes a data value that \( S_i \) uses.
- It implies that \( S_i \) must execute before \( S_j \). 

\[
\begin{align*}
S \delta S_i & \quad (S \delta S_i \quad \text{and} \quad S \delta S_j)
\end{align*}
\]
**Data Dependence**

1. $S_1: A = 1.0$
2. $S_2: B = A + 2.0$
3. $S_3: A = C - D$
4. $S_4: A = B/C$

We define four types of data dependence.

- **Input dependence**: a statement $S_i$ precedes a statement $S_j$ in execution and $S_i$ uses a data value that $S_j$ also uses.

- Does this imply that $S_i$ must execute before $S_j$?

$S_i \in S_j \quad (S_i \in S_j)$

**Data Dependence (continued)**

- The dependence is said to **flow** from $S_i$ to $S_j$ because $S_i$ precedes $S_j$ in execution.

- $S_i$ is said to be the **source** of the dependence. $S_j$ is said to be the **sink** of the dependence.

- The only "true" dependence is flow dependence; it represents the flow of data in the program.

- The other types of dependence are caused by programming style; they may be eliminated by re-naming.

$S_1: A = 1.0$
$S_2: B = A + 2.0$
$S_3: A1 = C - D$
$S_4: A2 = B/C$

**Value or Location?**

- There are two ways a dependence is defined: **value-oriented** or **location-oriented**.

$S_1: A = 1.0$
$S_2: B = A + 2.0$
$S_3: A = C - D$
$S_4: A = B/C$
Example 1

for i = 2 to 4 {
    S\textsuperscript{1}: a[i] = b[i] + c[i] ;
    S\textsuperscript{2}: d[i] = a[i]
}

- There is an instance of S\textsuperscript{1} that precedes an instance of S\textsuperscript{2} in execution and S\textsuperscript{1} produces data that S\textsuperscript{2} consumes.
- S\textsuperscript{1} is the source of the dependence; S\textsuperscript{2} is the sink of the dependence.
- The dependence flows between instances of statements in the same iteration (loop-independent dependence).
- The number of iterations between source and sink (dependence distance) is 0. The dependence direction = .

Example 2

do i = 2, 4
    S\textsuperscript{1}: a(i) = b(i) + c(i)
    S\textsuperscript{2}: d(i) = a(i-1)
end do

- There is an instance of S\textsuperscript{1} that precedes an instance of S\textsuperscript{2} in execution and S\textsuperscript{1} produces data that S\textsuperscript{2} consumes.
- S\textsuperscript{1} is the source of the dependence; S\textsuperscript{2} is the sink of the dependence.
- The dependence flows between instances of statements in different iterations (loop-carried dependence).
- The dependence distance is 1. The direction is positive ().

Example 3

do i = 2, 4
    S\textsuperscript{1}: a(i) = b(i) + c(i)
    S\textsuperscript{2}: d(i) = a(i+1)
end do

- There is an instance of S\textsuperscript{2} that precedes an instance of S\textsuperscript{1} in execution and S\textsuperscript{2} consumes data that S\textsuperscript{1} produces.
- S\textsuperscript{2} is the source of the dependence; S\textsuperscript{1} is the sink of the dependence.
- The dependence is loop-carried.
- The dependence distance is 1.
- Are you sure you know why it is even though S\textsuperscript{1} appears before S\textsuperscript{2} in the code?

Example 4

do i = 2, 4
do j = 2, 4
    S: a(i,j) = a(i-1,j+1)
end do
end do

- An instance of S precedes another instance of S and S produces data that S consumes.
- S is both source and sink.
- The dependence is loop-carried.
- The dependence distance is (1,-1).
- S δ\textsuperscript{1} S \textsuperscript{1}
- S δ\textsuperscript{2} S \textsuperscript{2}
- S δ\textsuperscript{3} S \textsuperscript{3}
- S δ\textsuperscript{4} S \textsuperscript{4}
### Problem Formulation
- Consider the following **perfect** nest of depth $d$:

```latex
\begin{align*}
\text{do } i_1 = L_1, U_1 \\
\text{do } i_2 = L_2, U_2 \\
\ldots \\
\text{do } i_d = L_d, U_d \\
\text{enddo}
\end{align*}
```

The dependence direction vector is $\text{sign}(1) = \text{array reference}$.

The dependence distance vector is $i_j$.

### Problem Formulation - Example
- Does there exist two iteration vectors $i_1$ and $i_2$, such that $2 \leq i_1 \leq i_2 \leq 4$ and such that:
  
  $i_1 = i_2 - 1$?

- Answer: yes; $i_1 = 2$ & $i_2 = 3$ and $i_1 = 3$ & $i_2 = 4$.

- Hence, there is dependence!

- The dependence distance vector is $i_2 - i_1 = 1$.

- The dependence direction vector is $\text{sign}(1) = \text{something}$. 

### Problem Formulation - Example
- Dependence will exist if there exists two iteration vectors $i$ and $j$ such that $L \leq k \leq U$ and:
  
  $f_k(i) = g_k(j)$

  and

  $f_k(i) = g_k(j)$

  and

  $f_k(i) = g_k(j)$

  and

  $f_k(i) = g_k(j)$

  and

  $f_k(i) = g_k(j)$

- That is:
  
  $f_k(i) - g_k(j) = 0$

  and

  $f_k(i) - g_k(j) = 0$

  and

  $f_k(i) - g_k(j) = 0$

  and

  $f_k(i) - g_k(j) = 0$

  and

  $f_k(i) - g_k(j) = 0$
Problem Formulation - Example

\begin{verbatim}
do i = 1, 10
   S1: a(2*i) = b(i) + c(i)
   S2: d(i) = a(2*i+1)
end do
\end{verbatim}

- Does there exist two iteration vectors $i_1$ and $i_2$, such that $1 \leq i_1 \leq i_2 \leq 10$ and such that:
  \[ 2*i_1 = 2*i_2 + 1? \]
- Answer: no; $2*i_1$ is even & $2*i_2+1$ is odd.
- Hence, there is no dependence!

Problem Formulation

- Dependence testing is equivalent to an integer linear programming (ILP) problem of 2d variables & m+d constraint!
- An algorithm that determines if there exists two iteration vectors $k$ and $j$ that satisfies these constraints is called a dependence tester.

\begin{verbatim}
do k = L_k, U_k
   do j = L_j, U_j
      do I = L_I, U_I
         a(I) = a(I)
         d(I) = a(I)
      enddo
   enddo
enddo
\end{verbatim}

- The dependence distance vector is given by $j - k$.
- The dependence direction vector is given by $\text{sign}(j - k)$.
- Dependence testing is NP-complete!
- A dependence test that reports dependence only when there is dependence is said to be exact. Otherwise it is in-exact.
- A dependence test must be conservative; if the existence of dependence cannot be ascertained, dependence must be assumed.

Dependence Testers

- Lamport's Test.
- GCD Test.
- Banerjee's Inequalities.
- Generalized GCD Test.
- Power Test.
- I-Test.
- Omega Test.
- Delta Test.
- Stanford Test.
- etc...
Lamport's Test

- Lamport's Test is used when there is a single index variable in the subscript expressions, and when the coefficients of the index variable in both expressions are the same.

\[ A(\ldots, b^i + c_1, \ldots) = \ldots = A(\ldots, b^i + c_2, \ldots) \]

- The dependence problem: does there exist \( i_1 \) and \( i_2 \), such that
  \[ L_i \leq i_1 \leq i_2 \leq U_i \]
  and such that
  \[ b \cdot i_1 + c_1 = b \cdot i_2 + c_2 \]

- There is integer solution if and only if \( \frac{c_1 - c_2}{b} \) is integer.

- The dependence distance is \( d = \frac{c_1 - c_2}{b} \) if \( I_i \leq |d| \leq U_i \).

- \( d > 0 \) \( \Rightarrow \) true dependence.
- \( d = 0 \) \( \Rightarrow \) loop independent dependence.
- \( d < 0 \) \( \Rightarrow \) anti dependence.

\[ i_1 = i_2 - 1? \]

\[ b = 1; c_1 = 0; c_2 = -1 \]

\[ \frac{c_1 - c_2}{b} = 1 \]

There is dependence. Distance \( (i) \) is 1.

\[ j_1 = j_2 + 1? \]

\[ b = 1; c_1 = 0; c_2 = 1 \]

\[ \frac{c_1 - c_2}{b} = -1 \]

There is dependence. Distance \( (j) \) is -1.

GCD Test

- Given the following equation:

\[ \sum_{i=1}^{n} a_i x_i = c \]

Where \( a_i \) and \( c \) are integers

An integer solution exists if and only if:

\[ \text{gcd}(a_1, a_2, \ldots, a_n) \text{ divides } c \]

- Problems:
  - ignores loop bounds
  - gives no information on distance or direction of dependence
  - often \( \text{gcd}(\ldots) \) is 1 which always divides \( c \), resulting in false dependences
GCD Test - Example

\begin{align*}
do \ i = 1, 10 \\
S_1: \ & a(2^i) = b(i) + c(i) \\
S_2: \ & d(i) = a(2^i - 1) \\
end \ do
\end{align*}

- Does there exist two iteration vectors \( i_1 \) and \( i_2 \), such that \( 1 \leq i_1 \leq i_2 \leq 10 \) and such that:
  - \( 2^*i_1 = 2^*i_2 - 1 \)?
  - \( 2^*i_2 = 2^*i_1 - 1 \)?
- There will be an integer solution if and only if \( \gcd(2,-2) \) divides 1.
- This is not the case, and hence, there is no dependence!

GCD Test Example

\begin{align*}
do \ i = 1, 10 \\
S_1: \ & a(i) = b(i) + c(i) \\
S_2: \ & d(i) = a(i-100) \\
end \ do
\end{align*}

- Does there exist two iteration vectors \( i_1 \) and \( i_2 \), such that \( 1 \leq i_1 \leq i_2 \leq 10 \) and such that:
  - \( i_1 = i_2 - 100 \)?
  - \( i_2 - i_1 = 100 \)?
- There will be an integer solution if and only if \( \gcd(1,-1) \) divides 100.
- This is the case, and hence, there is dependence! Or is there?

Dependence Testing Complications

- Unknown loop bounds:
  \begin{align*}
  \text{do } i = 1, N \\
  S_1: \ & a(i) = a(i+10) \\
  end \ do
  \end{align*}
  What is the relationship between \( N \) and 10?

- Triangular loops:
  \begin{align*}
  \text{do } i = 1, N \\
  \text{do } j = 1, i-1 \\
  S: \ & a(j,j) = a(j,j) \\
  end \ do \\
  end \ do
  \end{align*}
  Must impose \( j < i \) as an additional constraint.

More Complications

- User variables:
  \begin{align*}
  \text{do } i = 1, 10 \\
  S_1: \ & a(i) = a(i+k) \\
  end \ do
  \end{align*}
  Same problem as unknown loop bounds, but occur due to some loop transformations (e.g., loop bounds normalization).
  \begin{align*}
  \text{do } i = L, H \\
  S_1: \ & a(i) = a(i-1) \\
  end \ do
  \end{align*}
  \begin{align*}
  \text{do } i = 1, H-L \\
  S_1: \ & a(i+L) = a(i+L-1) \\
  end \ do
  \end{align*}
More Complications

- Scalars:
  - do $i = 1, N$
    - $S_1$: $x = a(i)$
    - $S_2$: $b(i) = x$
    - end do
  - $j = N-1$
    - do $i = 1, N$
      - $S_1$: $a(i) = a(j)$
      - $S_2$: $j = j - 1$
      - end do
  - sum = 0
    - do $i = 1, N$
      - $S_1$: sum + sum = $a(i)$
      - $S_2$: sum = sum(i) $i = 1, N$
      - end do

Optimizing Compilers: Parallelization

Loop Parallelization

- A dependence is said to be carried by a loop if the loop is the outermost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is loop-independent.

  - do $i = 2, n-1$
    - do $j = 2, m-1$
      - $a(i, j) = ...$
      - $b(i, j) = ...$
      - $c(i, j) = ...$
    - end do
  - end do

  The iterations of a loop may be executed in parallel with one another if and only if no dependences are carried by the loop!
Loop Parallelization - Example

- Iterations of loop j must be executed sequentially, but the iterations of loop i may be executed in parallel.

- Outer loop parallelism.

\[
\begin{aligned}
\delta_{i<} & \\
do & i = 2, n-1 \\
do & j = 2, m-1 \\
\quad & b(i, j) = \ldots \\
\quad & = b(i, j-1) \\
end & do \\
end & do
\end{aligned}
\]

\[
\begin{aligned}
\delta_{i>_<} & \\
do & i = 2, n-1 \\
do & j = 2, m-1 \\
\quad & b(i, j) = \ldots \\
\quad & = b(i-1, j) \\
end & do \\
end & do
\end{aligned}
\]

Loop Interchange

Loop interchange changes the order of the loops to improve the spatial locality of a program.

\[
\begin{aligned}
do & j = 1, n \\
do & i = 1, n \\
\quad & a(i,j) \ldots \\
end & do \\
end & do
\end{aligned}
\]
**Loop Interchange**

Loop interchange changes the order of the loops to improve the spatial locality of a program.

\[
\begin{align*}
&\text{do } i = 1, n \\
&\quad \text{do } j = 1, n \\
&\quad \quad \ldots a(i,j) \ldots \\
&\quad \quad \ldots a(i,j) \ldots \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]

\[
\begin{align*}
&\text{do } i = 1, n \\
&\quad \text{do } j = 1, n \\
&\quad \quad \ldots a(i,j) \ldots \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]

- Loop interchange can improve the granularity of parallelism!

\[
\begin{align*}
&\text{do } i = 1, n \\
&\quad \text{do } j = 1, n \\
&\quad \quad a(i,j) = b(i,j) \\
&\quad \quad c(i,j) = a(i-1,j) \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]

- When is loop interchange legal?
Loop Interchange

When is loop interchange legal?

When the "interchanged" dependences remain lexicographically positive!

Loop Blocking (Tiling)

When is loop blocking legal?

Wednesday's Class

Global Scheduling, Software Pipelining [ALSU 10.4 - 10.5]