Data Dependence, Parallelization, and Locality Enhancement

(courtesy of Tarek Abdelrahman, University of Toronto)

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 $S_j$.

- **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 $S_j$.

- **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 $S_j$.

\begin{align*}
S_1 & : A = 1.0 \\
S_2 & : B = A + 2.0 \\
S_3 & : A = C - D \\
\vdots & \\
S_k & : A = B / C
\end{align*}

\begin{align*}
S_1 & : A = 1.0 \\
S_2 & : B = A + 2.0 \\
S_3 & : A = C - D \\
\vdots & \\
S_k & : A = B / C
\end{align*}

We define four types of data dependence.
Data Dependence

\[ S_i : \quad A = 1.0 \]
\[ S_2 : \quad B = A + 2.0 \]
\[ S_3 : \quad A = C - D \]
\[ S_4 : \quad A = B/C \]

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 \( S_j \)?

\[ S_i \delta S_j \quad (S_i \delta S_j) \]

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.

\[ S_1 : \quad A = 1.0 \]
\[ S_2 : \quad B = A + 2.0 \]
\[ S_3 : \quad A = C - D \]
\[ S_4 : \quad A = B/C \]

Data Dependence (continued)

- Data dependence in a program may be represented using a dependence graph \( G=(V,E) \), where the nodes \( V \) represent statements in the program and the directed edges \( E \) represent dependence relations.

\[ S_1 : \quad A = 1.0 \]
\[ S_2 : \quad B = A + 2.0 \]
\[ S_3 : \quad A = C - D \]
\[ S_4 : \quad A = B/C \]

Value or Location?

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

\[ S_1 : \quad A = 1.0 \]
\[ S_2 : \quad B = A + 2.0 \]
\[ S_3 : \quad A = C - D \]
\[ S_4 : \quad A = B/C \]
Example 1

\[ \text{do } i = 2, 4 \]
\[ \quad S_1: a(i) = b(i) + c(i) \]
\[ \quad \vdots \]
\[ \quad S_2: d(i) = a(i) \]
\[ \text{end do} \]

- There is an instance of \( S_1 \) that precedes an instance of \( S_2 \) in execution and \( S_1 \) produces data that \( S_2 \) consumes.
- \( S_1 \) is the source of the dependence; \( S_2 \) 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 is =.

Example 2

\[ \text{do } i = 2, 4 \]
\[ \quad S_1: a(i) = b(i) + c(i) \]
\[ \quad \vdots \]
\[ \quad S_2: d(i) = a(i-1) \]
\[ \text{end do} \]

- There is an instance of \( S_1 \) that precedes an instance of \( S_2 \) in execution and \( S_1 \) produces data that \( S_2 \) consumes.
- \( S_1 \) is the source of the dependence; \( S_2 \) 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

\[ \text{do } i = 2, 4 \]
\[ \quad S_1: a(i) = b(i) + c(i) \]
\[ \quad \vdots \]
\[ \quad S_2: d(i) = a(i+1) \]
\[ \text{end do} \]

- There is an instance of \( S_2 \) that precedes an instance of \( S_1 \) in execution and \( S_2 \) consumes data that \( S_1 \) produces.
- \( S_2 \) is the source of the dependence; \( S_1 \) is the sink of the dependence.
- The dependence is loop-carried.
- The dependence distance is 1.
- \[ S_2 \delta^i S_1 \quad \text{or} \quad S_1 \delta^i S_2 \]

Example 4

\[ \text{do } i = 2, 4 \]
\[ \quad \text{do } j = 2, 4 \]
\[ \quad S: a(i,j) = a(i-1,j+1) \]
\[ \text{end do} \]
\[ \text{end do} \]

- 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 \((1,-1)\).
- \[ S \delta^i_{(1,-1)} S \quad \text{or} \quad S \delta^i_{(-1,1)} S \]
Problem Formulation

- Consider the following perfect nest of depth d:

\[
\begin{align*}
\text{do } i_1 = L_{i_1}, U_{i_1} \\
\text{do } i_2 = L_{i_2}, U_{i_2} \\
\quad \text{do } i_d = L_{i_d}, U_{i_d} \\
\quad \quad \text{do } j = L_j, U_j \\
\quad \quad \quad \text{do } k = L_k, U_k \\
\quad \quad \quad \quad a(f(I)), I(a(f(I))) = \ldots \\
\quad \quad \quad \quad \ldots = a(g(I)), I(a(g(I))) \\
\quad \quad \quad \quad \text{endo} \\
\quad \quad \quad \text{endo} \\
\quad \quad \text{endo} \\
\quad \text{endo} \\
\text{endo} \\
\text{endo} \\
\text{endo} \\
\end{align*}
\]

- That is:

\[
\begin{align*}
f_i(k) &= g_i(j) \\
f_2(k) &= g_2(j) \\
\vdots \\
f_m(k) &= g_m(j) \\
\end{align*}
\]

Does there exist two iteration vectors \( i_1 \) and \( i_2 \), such that

\[
2 \leq i_1 \leq i_2 \leq 4 \text{ and such that:}
\]

- \( i_1 = i_2 - 1? \)
- 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 sign(1) = <.

Problem Formulation - Example

- Do i = 2, 4
  \[
  \begin{align*}
  S_1: & \quad a(i) = b(i) + c(i) \\
  S_2: & \quad d(i) = d(i-1) 
  \end{align*}
  \]

- Does there exist two iteration vectors \( i_1 \) and \( i_2 \), such that

\[
2 \leq i_1 \leq i_2 \leq 4 \text{ and such that:}
\]

- \( i_1 = i_2 + 1? \)
- Answer: yes; \( i_1 = 3 \) & \( i_2 = 2 \) and \( i_1 = 4 \) & \( i_2 = 3 \). (But, but!).
- Hence, there is dependence!
- The dependence distance vector is \( i_2 - i_1 = -1 \).
- The dependence direction vector is sign(-1) = >.
- Is this possible?
Problem Formulation - Example

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

- 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:
  \[ 2i_1 = 2i_2 + 1? \]
  - Answer: no; \( 2i_1 \) is even & \( 2i_2 + 1 \) 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 exits two iteration vectors \( k \) and \( j \) that satisfies these constraints is called a dependence tester.
  - 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.
  \[
  A(\ldots, b^{*}i + c_1, \ldots) = \ldots \\
  \ldots = A(\ldots, b^{*}i + c_2, \ldots)
  \]
  - 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
    \[ b^{*}i_1 + c_1 = b^{*}i_2 + c_2 \quad \text{or} \quad i_2 - i_1 = \frac{c_1 - c_2}{b} ? \]
  - 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 \( L_i \leq |d| \leq U_i \).
  - \( d > 0 \) \( \Rightarrow \) true dependence.
  - \( d = 0 \) \( \Rightarrow \) loop independent dependence.
  - \( d < 0 \) \( \Rightarrow \) anti dependence.
Lamport's Test - Example

\[
\begin{align*}
d & \text{do } i = 1, n \\
d & \text{do } j = 1, n \\
S & : a(i, j) = a(i-1, j+1) \\
& \text{end do} \\
& \text{end do}
\end{align*}
\]

- \( i_1 = i_2 - 1? \)
- \( b = 1; c_1 = 0; c_2 = -1 \)
- \( c_1 - c_2 = 1 \)
- \( b \)
- There is dependence.
- Distance (i) is 1.

\[
\begin{align*}
d & \text{do } i = 1, n \\
d & \text{do } j = 1, n \\
S & : a(2i, j) = a(i-1, 2j+1) \\
& \text{end do} \\
& \text{end do}
\end{align*}
\]

- \( j_1 = j_2 + 1? \)
- \( b = 1; c_1 = 0; c_2 = 1 \)
- \( c_1 - c_2 = 1 \)
- \( b \)
- There is no dependence.
- Distance (j) is -1.

GCD Test

- Given the following equation:
  \[
  \sum_{i=1}^{n} a_i x_i = c
  \]
  \( a_i \)'s and c are integers

  an integer solution exists if and only if:
  \[
gcd(a_1, a_2, \cdots, a_n) \text{ divides } c
  \]

- Problems:
  - ignores loop bounds.
  - gives no information on distance or direction of dependence.
  - often gcd(\( \cdots \)) is 1 which always divides c, resulting in false dependences.

GCD Test - Example

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

- 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:
  \[
  2^i_1 = 2^i_2 - 1? \\
  \text{or} \\
  2^i_2 - 2^i_1 = 1?
  \]

- 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**

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

- 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:
  \( i_1 = i_2 - 100 ? \)  
  or \( i_2 - i_1 = 100 ? \)
- There will be an integer solution if and only if \( \text{gcd}(1, -1) \) divides 100.
- This is the case, and hence, there is dependence! Or is there?

---

**Dependence Testing Complications**

- **Unknown loop bounds.**
  ```
  do i = 1, N
  S1: a(i) = a(i+10)
  end do
  ```
  What is the relationship between \( N \) and 10?

- **Triangular loops.**
  ```
  do i = 1, N
  do j = 1, i-1
  S: a(i, j) = a(j, i)
  end do
  end do
  ```
  Must impose \( j < i \) as an additional constraint.

---

**More Complications**

- **User variables.**
  ```
  do i = 1, 10
  S1: a(i) = a(i+k)
  end do
  ```
  Some problem as unknown loop bounds, but occur due to some loop transformations (e.g., normalization).

- **Scalars.**
  ```
  do i = 1, N
  S1: x(i) = a(i)
  S2: b(i) = x
  end do
  ```
  ```
  do i = 1, N
  S1: a(i) = a(N-i)
  end do
  ```
  ```
  sum = 0
  do i = 1, N
  S1: sum = sum + a(i)
  end do
  ```
  ```
  do i = 1, N
  S1: sum += sum + a(i)
  sum += sum(i)
  ```
  ```
  do i = 1, N
  S1: x(i) = a(i)
  S2: b(i) = x
  end do
  ```
  Must impose \( j < i \) as an additional constraint.

---

**More Complications**

- **User variables.**
  ```
  do i = 1, 10
  S1: a(i) = a(i+k)
  end do
  ```
  Some problem as unknown loop bounds, but occur due to some loop transformations (e.g., normalization).

- **Scalars.**
  ```
  do i = 1, N
  S1: x(i) = a(i)
  S2: b(i) = x
  end do
  ```
  ```
  do i = 1, N
  S1: a(i) = a(N-i)
  end do
  ```
  ```
  sum = 0
  do i = 1, N
  S1: sum = sum + a(i)
  end do
  ```
  ```
  do i = 1, N
  S1: sum += sum + a(i)
  sum += sum(i)
  ```
  ```
  do i = 1, N
  S1: x(i) = a(i)
  S2: b(i) = x
  end do
  ```
  Must impose \( j < i \) as an additional constraint.
### Serious Complications

- **Aliases.**
  - Equivalence Statements in Fortran:
    
    ```fortran
    real a(10,10), b(10)
    makes b the same as the first column of a.
    ```
  - Common blocks: Fortran's way of having shared/global variables.
    ```fortran
    common /shared/a,b,c
    subroutine foo (...) common /shared/a,b,c
    common /shared/x,y,z
    ```

### Loop Parallelization

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

```fortran
do i = 2, n-1
  do j = 2, m-1
    a(i, j) = ...
    ... = a(i, j)
    b(i, j) = ...
    ... = b(i, j-1)
    c(i, j) = ...
    ... = c(i-1, j)
  end do
end do
```
Loop Parallelization

A dependence is said to be carried by a loop if the loop is the outmost 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) = ... = a(i, j)
    b(i, j) = ... = b(i, j-1)
    c(i, j) = ... = c(i-1, 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!

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

Outer loop parallelism.
Loop Parallelization - Example

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

Loop Interchange

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

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

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

When is loop interchange legal? when the "interchanged" dependences remain lexiographically positive!

Loop Blocking (Loop Tiling)
Exploits temporal locality in a loop nest.

\[
\begin{align*}
d & = 1, T \\
i & = 1, n \\
j & = 1, n \\
\text{... } a(i,j) \\
\text{... } & \\
\text{end do} \\
\text{end do}
\end{align*}
\]
Loop Blocking (Loop Tiling)
Exploits temporal locality in a loop nest.

\[ \text{do ic = 1, n, B} \]
\[ \text{do jc = 1, n, B} \]
\[ \text{do } t = 1, T \]
\[ \text{do } i = 1, B \]
\[ \text{do } j = 1, B \]
\[ \ldots a(i(i-1), j(c+j-1)) \]
\[ \text{end do} \]
\[ \text{end do} \]
\[ B: \text{ Block size} \]

When is loop blocking legal?