Dependence Analysis

Last Time:

• Brief introduction to interprocedural analysis

Today:

Optimization for parallel machines and memory hierarchies

• Dependence analysis
• Loop transformations
• an example - McKinley, Carr, Tseng loop transformations to improve cache performance

After that:

• TRIPS Architecture and Compiler (scheduling)

Dependence Examples

Can either of these loops be performed in parallel?

A loop-independent dependence exists regardless of the loop structure. They do not inhibit parallelization, but they do affect statement order which with a loop.

A loop-carried dependence is induced by the iterations of a loop and prevents safe loop parallelization.
Dependence Classification

\( S_1 \delta S_2 \)

**True (flow) dependence**
occurs when \( S_1 \) writes a memory location that \( S_2 \) later reads.

**Anti dependence**
occurs when \( S_1 \) reads a memory location that \( S_2 \) later writes.

**Output dependence**
occurs when \( S_1 \) writes a memory location that \( S_2 \) later writes.

**Input dependence**
occurs when \( S_1 \) reads a memory location that \( S_2 \) later reads. (Input dependences do not restrict statement order.)

Dependence Analysis Question

Given

\[
\begin{align*}
\text{DO } i_1 &= L_1, U_1 \\
&\quad \ldots \text{ DO } i_n = L_n, U_n \\
S_1 &\quad A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots \\
S_2 &\quad \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))
\end{align*}
\]

A *dependence* between statement \( S_1 \) and \( S_2 \), denoted \( S_1 \delta S_2 \), indicates that \( S_1 \), the *source*, must be executed before \( S_2 \), the *sink* on some iteration of the nest.

Let \( \alpha \& \beta \) be a vector of \( n \) integers within the ranges of the lower and upper bounds of the \( n \) loops.

**Does** \( \exists \alpha \leq \beta \), s.t.

\[
f_k(\alpha) = g_k(\beta) \quad \forall k, \ 1 \leq k \leq m
\]


**Iteration Space**

$\text{do } I = 1, 5$
$\text{do } J = I, 6$
$\ldots$
$\text{enddo}$
$\text{enddo}$

$1 \leq I \leq 5$
$I \leq J \leq 6$

- Lexicographical (sequential) order
  
  $(1,1), (1,2), \ldots, (1,6)$
  $(2,1), (2,2), \ldots, (2,6)$
  \ldots
  $(5,1), (5,2), \ldots, (5,6)$

- Given $I = (i_1, \ldots, i_n)$ and $I' = (i'_1, \ldots, i'_n)$,
  $I < I'$ iff
  
  $(i_1, i_2, \ldots, i_k) = (i'_1, i'_2, \ldots, i'_k)$ & $i_{k+1} < i'_{k+1}$

**Distance & Direction Vectors**

$\text{do } I = 1, N$
$\text{do } J = 1, N$
$S_1 \quad A(I,J) = A(I,J-1) + 1$
$\text{enddo}$
$\text{enddo}$

$\text{do } I = 1, N$
$\text{do } J = 1, N$
$S_2 \quad A(I,J) = A(I-1,J-1) + 1$
$\text{enddo}$
$\text{enddo}$

$\text{do } I = 1, N$
$\text{do } J = 1, N$
$S_3 \quad B(I,J) = B(I-1,J+1) + 1$
$\text{enddo}$
$\text{enddo}$

**Distance Vector:** number of iterations between accesses

**Direction Vector:** direction in the iteration space

distance vector direction vector

$S_1 \delta S_1$
$S_2 \delta S_2$
$S_3 \delta S_3$
Which Loops are Parallel?

\[
\begin{align*}
S_1 & \quad A(I,J) = A(I,J-1) + 1 \\
S_2 & \quad A(I,J) = A(I-1,J-1) + 1 \\
S_3 & \quad B(I,J) = B(I-1,J+1) + 1
\end{align*}
\]

- A dependence \( D = (d_1, \ldots, d_k) \) is carried at level \( i \), if \( d_i \) is the first nonzero element of the distance/direction vector.
- A loop \( l_i \) is parallel, if \( \not\exists \) a dependence \( D_j \) carried at level \( i \). Either

| \( \forall D_j \) | distance vector | \( d_1, \ldots, d_{i-1} > 0 \) | direction vector | \( d_1, \ldots, d_{i-1} = "<" \) |
| --- | --- | --- | --- |
| OR | \( d_1, \ldots, d_i = 0 \) | \( d_1, \ldots, d_i = "=" \) |

Approaches to Dependence Testing

- Can we solve this problem exactly?
- What is conservative in this framework?
- Restrict the problem to consider index and bound expressions that are linear functions

\[ \Rightarrow \text{solve general system of linear equations} \]

NP-complete

Solution Methods

- Cascade of exact, efficient tests (if they fail, use inexact methods)
  - Rice
  - Stanford
- Inexact methods
  - GCD
  - Banerjee’s inequalities (Illinois)
  - Fourier-Motzkin (Pugh)
Greatest Common Denominator (GCD) - Inexact test

\[
\begin{align*}
\text{do } i &= 1, N \\
\text{a}(2i+1) &= \text{a}(8i+3) + \text{a}(4i) \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
f(I) &= 2i+1 \\
g(I') &= 4i'
\end{align*}
\]

let 
\[
\begin{align*}
f(I) &= \alpha_0 + \alpha_1 i_1 + \ldots + \alpha_k i_k \\
g(I') &= \beta_0 + \beta_1 i'_1 + \ldots + \beta_k i'_k
\end{align*}
\]

- Test for integer solutions to \( f(I) = g(I') \)
  \[
  \alpha_1 i_1 - \beta_1 i'_1 + \ldots + \alpha_k i_k - \beta_k i'_k = \alpha_0 - \beta_0
  \]

- \( \exists \) a solution \( \text{iff} \)
  \[
  \gcd(\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k) = |\alpha_0 - \beta_0|
  \]

- If the \( \gcd = 1 \), what do we know?

- If the \( \gcd > 1 \), we test to determine if the index expression ranges over that value, if so \( \implies \exists \) a dependence.

Banerjee - Inexact test

- Tests for a real solution to the integer equations
- For example, given a single index variable in the subscripts (e.g., \( 2i \) and \( i+3 \)) determines if the lines intersect at a real or integer point.

\[
\begin{align*}
\text{let } h(I, I') &= \alpha I - \beta I' \\
\text{max}_R h(I_k, I'_k) &= \max_R h(I_k, I'_k) \\
\text{min}_R h(I_k, I'_k) &= \min_R h(I_k, I'_k)
\end{align*}
\]

\( I_k D I'_k \) is the relation imposed by the direction vector element (either \(<, >, \) or \(=\))

Banerjee's inequality

- For a given direction vector \( D \), \( \exists \) a real solution to
  \[
  \sum_{i=1}^n H_i - D_i \leq \beta_0 - \alpha_0 \leq \sum_{i=1}^n H_i^+ + D_i
  \]

Direction Vector Hierarchy:

\[
\begin{align*}
(\ast, \ast) \\
(\ast, \ast) \quad (\ast, \ast) \\
(\ast, \ast) \\
(\ast, \ast) \quad (\ast, \ast) \quad (\ast, \ast)
\end{align*}
\]
Exact Test Cascade

- Stanford [Maydan, Hennessy, Lam - PLDI '91]
  - Single variable per constraint: each constraint can be solved directly
  - Acyclic test: variable is constrained by other variables in only one direction, replace variable with lower (upper) bound
  - Loop Residue Test: each constraint is of the form $i - i' \leq \alpha$, cycle with a negative value implies dependence
  - Fourier-Motzkin (inexact)

- Rice [Goff, Kennedy, Tseng - PLDI '91]
  - Index variable classification (complexity & separability)
  - ZIV test, Strong and weak SIV tests
  - Delta test for coupled subscripts: propagate constraints from separable subscripts to determine independence
  - MIV - Banerjee (inexact)

Subscript Classification

Complexity:

- $A(1, i+1, j)$
- $A(N, i, i)$

Classification by the number of index variables occurring in subscript

- ZIV $\rightarrow$ zero index variable (51%)
- SIV $\rightarrow$ single index variable (46%)
- MIV $\rightarrow$ multiple index variable (3%)

Separability:

- $A(i+1, j, j)$
- $A( i, j, k)$

Classification by determining if index variables are shared in subscripts

- Separable (Allen '83)
  - Each subscript expression has disjoint index variables
- Coupled (Li, Yew, Zhu '89)
  - Subscripts expressions share index variables
Taking Advantage of Separability

Separable subscripts

- may be tested independently
- merge the resulting dependence information

Direction Vector Hierarchy

Partition Based

\[
\begin{align*}
\langle *, * \rangle & \quad \langle *, * \rangle \\
\langle <, * \rangle & \quad \langle <, * \rangle \\
\langle <, > \rangle & \quad \langle <, > \rangle \\
\langle <, = \rangle & \quad \langle <, = \rangle \\
\langle >, * \rangle & \quad \langle >, * \rangle \\
\langle >, = \rangle & \quad \langle >, = \rangle \\
\langle =, * \rangle & \quad \langle =, * \rangle \\
\langle =, = \rangle & \quad \langle =, = \rangle \\
\end{align*}
\]

Partition-Based Algorithm:

1. Partition into separable & coupled groups
2. Classify as ZIV, SIV, MIV subscripts
3. Apply dependence tests to each group
4. Finished if independent
5. Otherwise merge dependence information

ZIV test

Example: test \( A(e_1) \) & \( A(e_2) \)

Algorithm:

- if \( e_1 \neq e_2 \) then independent

Symbolic test:

- symbolically compute \( e_1 - e_2 \)
**SIV Subscripts**

Test \( A(a_1 I + c_1) \) & \( A(a_2 I + c_2) \)

**Strong SIV** \( (a_1 = a_2) \)

Algorithm:
- distance \( d = (c_1 - c_2) / a \)
- independent if
  1. \( d \) is not integer, OR
  2. \( |d| > U - L \)

Symbolic test:
- symbolically compute \( c_1 - c_2 \)
- symbolically compare \( d, U, L \)

**Weak SIV** \( (a_1 = 0 \text{ or } a_2 = 0) \)

**Crossing SIV** \( (a_1 = -a_2) \)

---

**Delta Test**

- Multiple subscript test
  - Exact for common coupled subscripts
- Constraints for index variable
  - Derived from SIV subscripts
  - Distance, line, point
  - Intersect/propagate → other subscripts

**Constraint Intersection**

Example: test \( A(I, I) \) & \( A(I+1, I+2) \)
Constraints must hold simultaneously (intersection)

\[
  c_1 \cap c_2 = \{d_1 = 1\} \cap \{d_2 = 2\} = \emptyset
\]

⇒ no intersection proves independence

**Constraint Propagation**

Example: test \( A(I+1, I+J) \) & \( A(I, I+J) \)

Propagate \( C_1 = \{d_1 = 1\} \) into second subscript

⇒ \( A(\ldots, J-1) \) & \( A(\ldots, J) \)
⇒ Generate \( C_2 = \{d_2 = -1\} \)
⇒ distance vector \( (1, -1) \)
**Empirical Study**

Programs

- Riceps, Perfect, Spec, Eispack, Linpack

Array reference pairs tested

- All reference pairs in loop nest
- After symbolic analysis phase
- Using symbolic expression simplifier

**Effectiveness**

<table>
<thead>
<tr>
<th>% of</th>
<th>ZIV</th>
<th>Strong SIV</th>
<th>Weak SIV</th>
<th>MIV</th>
<th>Delta</th>
<th>Sym Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>all subscripts</td>
<td>51</td>
<td>39</td>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all successful</td>
<td>31</td>
<td>52</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>all independ.</td>
<td>85</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>successful</td>
<td>44</td>
<td>97</td>
<td>90</td>
<td>58</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>independent</td>
<td>44</td>
<td>3</td>
<td>6</td>
<td>22</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

**Multiple Subscript Tests**

- Coupled subscripts
  - 20% of subscripts were coupled
  - 75% of coupled subscripts in Eispack
- Delta test
  - tested 82% with constraint intersection
  - tested 4% with constraint propagation
Summary

- Classifying subscripts is important
  - Complexity → fast exact tests
  - Separability → solve simple systems
- Real programs
  - Have simple subscripts
  - Simple tests are usually exact
- More practical to use quick exact tests
  - Dependence analysis for scalar compilers
  - Save the more powerful but expensive tests

Uses for Dependence Analysis

- parallelization (detection and optimization)
- vectorization
- loop optimizations
- instruction scheduling (pipelined and super scalar)
- cache optimizations

Next Time

Read: