Data Dependence Analysis

Data Dependence Analysis I

- Data dependence analysis determines what the constraints are on how a piece of code can be reorganized.
- If we can determine that no data dependencies exist between the different iterations of a loop we may be able to run the loop in parallel or transform it to make better use of the cache.
- For the code below, a compiler could determine that statement $S_1$ must execute before $S_2$, and $S_3$ before $S_4$. $S_2$ and $S_3$ can be executed in any order:

$S_1$: $A := 0$

$S_2$: $B := A$

$S_3$: $C := A + D$

$S_4$: $D := 2$

Dependence Graphs I

- There can be three kinds of dependencies between statements:
  - **Flow dependence**: (i) $\rightarrow$ (j)
  - Also, true dependence or definition-use dependence.

(i) $X := \ldots$

(j) $\ldots := X$

- Statement (i) generates (defines) a value which is used by statement (j). We write (i) $\rightarrow$ (j).

- Also, use-definition dependence.

(i) $\ldots := X$

(j) $X := \ldots$
Dependence Graphs II

- Statement (i) uses a value overwritten by statement (j). We write (i)→(j).

Output-dependence

- Also, definition-definition dependence.

(i) \( X := \cdots \) 
(j) \( X := \cdots \)

- Statements (i) and (j) both assign to (define) the same variable. We write (i)→(j).

- Regardless of the type of dependence, if statement (j) depends on (i), then (i) has to be executed before (j).

Data Dependence Analysis I

The Dependence Graph:

\[
\begin{align*}
S_1: & \quad A := 0; \\
S_2: & \quad B := A; \\
S_3: & \quad C := A + D; \\
S_4: & \quad D := 2;
\end{align*}
\]

- In any program without loops, the dependence graph will be acyclic.
- Other common notations are
  
<table>
<thead>
<tr>
<th>Flow</th>
<th>→</th>
<th>( \equiv )</th>
<th>( \equiv \delta^f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti</td>
<td>→+</td>
<td>( \equiv )</td>
<td>( \equiv \delta^a )</td>
</tr>
<tr>
<td>Output</td>
<td>→◦</td>
<td>( \equiv )</td>
<td>( \equiv \delta^o )</td>
</tr>
</tbody>
</table>

Loop Fundamentals I

- We'll consider only perfect loop nests, where the only non-loop code is within the innermost loop:

\[
\begin{align*}
& \text{FOR } i_1 := 1 \text{ TO } n_1 \text{ DO} \\
& \quad \text{FOR } i_2 := 1 \text{ TO } n_2 \text{ DO} \\
& \quad \cdots \\
& \quad \text{FOR } i_k := 1 \text{ TO } n_k \text{ DO} \\
& \quad \quad \text{statements} \\
& \quad \text{ENDFOR} \\
& \quad \cdots \\
& \text{ENDFOR} \\
& \text{ENDFOR}
\end{align*}
\]

- The iteration-space of a loop nest is the set of iteration vectors \((k\)-tuples\): \(\{1,1,1,\cdots\},\cdots,\{n_1,n_2,\cdots,n_k\}\).
Loop Fundamentals II

```plaintext
FOR i := 1 TO 3 DO
  FOR j := 1 TO 4 DO
    statement
  ENDFOR
ENDFOR

Iteration-space: \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \\
                \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \\
                \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 3, 4 \rangle \}.

Represented graphically:
```

Loop Fundamentals III

```plaintext
FOR i := 1 TO 3 DO
  FOR j := 1 TO i + 1 DO
    statement
  ENDFOR
ENDFOR

Iteration-space: \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \\
                \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \\
                \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 3, 4 \rangle \}.

Represented graphically:
```

Loop Fundamentals IV

- The index vectors can be lexicographically ordered. 
  \langle 1, 1 \rangle \prec \langle 1, 2 \rangle means that iteration \langle 1, 1 \rangle precedes \langle 1, 2 \rangle.
- In the loop
  ```plaintext
  FOR i := 1 TO 3 DO
    FOR j := 1 TO 4 DO
      statement
    ENDFOR
  ENDFOR
  ```
  the following relations hold: \langle 1, 1 \rangle \prec \langle 1, 2 \rangle, \langle 1, 2 \rangle \prec \langle 1, 3 \rangle, \\
  \langle 1, 3 \rangle \prec \langle 1, 4 \rangle, \langle 1, 4 \rangle \prec \langle 2, 1 \rangle, \langle 2, 1 \rangle \prec \langle 2, 2 \rangle, \ldots, \langle 3, 3 \rangle \prec \langle 3, 4 \rangle.
- The iteration-space, then, is the lexicographic enumeration of 
  the index vectors. Confused yet?
Loop Transformations I

- The reason that we want to determine loop dependencies is to make sure that loop transformations that we want to perform are legal.
- For example, (for whatever reason) we might want to run a loop backwards:

```plaintext
FOR i := 1 TO 4 DO
ENDFOR
```

⇒

```plaintext
FOR i := 4 TO 1 BY -1 DO
ENDFOR
```

- The original array is:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Loop Transformations II

- After the original loop the array holds:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- After the transformed loop the array holds:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>15</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- It is clear that, in this case, reversing the loop is not a legal transformation. The reason is that there is a data dependence between the loop iterations.
- In the original loop $A[i]$ is read before it's assigned to, in the transformed loop $A[i]$ is assigned to before it's read.

Loop Transformations III

- The dependencies are easy to spot if we unroll the loop:


  ▲ Unroll
  FOR $i := 1$ TO $4$ DO
  ENDFOR

- Hence, in this loop

  FOR $i := 1$ TO $4$ DO
    $S_1$: $\cdots := A[i + 1]$
    $S_2$: $A[i] := \cdots$
  ENDFOR

  there's an anti-dependence from $S_1$ to $S_2$: $S_1 \rightarrow S_2$

- In this loop

  FOR $i := 1$ TO $4$ DO
    $S_1$: $A[i] := \cdots$
    $S_2$: $\cdots := A[i - 1]$
  ENDFOR

  there's a flow-dependence from $S_1$ to $S_2$: $S_1 \leftarrow S_2$
Loop Dependence Analysis

Are there dependencies between the statements in a loop, that stop us from transforming it? A general, 1-dim loop:

\[
\text{FOR } i := \text{From TO To DO} \\
S_1: A[f(i)] := \cdots \\
S_2: \cdots := A[g(i)] \\
\text{ENDFOR}
\]

- \(f(i)\) and \(g(i)\) are the expressions that index the array \(A\). They’re often of the form \(c_1 \times i + c_2\) (\(c_i\) are constants).
- There’s a flow dependence \(S_1 \rightarrow S_2\), if, for some values of \(I_d\) and \(I_u\), From \(\leq I_d\), \(I_u \leq\) To, \(I_d < I_u\), \(f(I_d) = g(I_u)\), i.e. the two index expressions are the same.
- \(I_d\) is the index for the definition (\(A[I_d]:=\cdots\)) and \(I_u\) the index for the use (\(\cdots:=A[I_u]\)).

Example

\[
\text{FOR } i := 1 \text{ TO 10 DO} \\
S_1: A[8 \times i + 3] := \cdots \\
S_2: \cdots := A[2 \times i + 1] \\
\text{ENDFOR}
\]

- \(f(I_d) = 8 \times I_d + 3\), \(g(I_u) = 2 \times I_u + 1\)
- Does there exist \(1 \leq I_d \leq 10\), \(1 \leq I_u \leq 10\), \(I_d < I_u\), such that \(8 \times I_d + 3 = 2 \times I_u + 1\)? If that’s the case, then \(S_1 \rightarrow S_2\).
- Yes, \(I_d = 1, I_u = 5 \Rightarrow 8 \times I_d + 3 = 11 = 2 \times I_u + 1\).
- There is a loop carried dependence between statement \(S_1\) and \(S_2\).
The GCD Test

Does there exist a dependence in this loop? I.e., do there exist integers \( I^d \) and \( I^u \), such that \( c \cdot I^d + j = d \cdot I^u + k \)?

\[
\begin{align*}
\text{FOR } I & := 1 \text{ TO } n \text{ DO} \\
S_1 & : \ A[c \cdot I + j] := \cdots \\
S_2 & : \cdots := A[d \cdot I + k] \\
\text{ENDFOR}
\end{align*}
\]

\( c \cdot I^d + j = d \cdot I^u + k \) only if gcd\((c, d)\) evenly divides \( k - j \), i.e. if \((k - j) \text{ mod gcd}(c, d) = 0 \).

This is a very simple and coarse test. For example, it doesn’t check the conditions \( 1 \leq I^d \leq n \), \( 1 \leq I^u \leq n \), \( I^d < I^u \).

There are many other much more exact (and complicated!) tests.

The GCD Test – Example I

\[
\begin{align*}
\text{FOR } I & := 1 \text{ TO } 10 \text{ DO} \\
S_1 & : \ A[2 \cdot I] := \cdots \\
S_2 & : \cdots := A[2 \cdot I + 1] \\
\text{ENDFOR}
\end{align*}
\]

\( c \cdot I^d + j = d \cdot I^u + k \) only if gcd\((c, d)\) evenly divides \( k - j \), i.e. if \((k - j) \text{ mod gcd}(c, d) = 0 \).

\( c = 2, j = 0, d = 2, k = 1 \).

\( (1 - 0) \text{ mod gcd}(2, 2) = 1 \text{ mod } 2 = 1 \Rightarrow S_1 \text{ and } S_2 \text{ are data independent!} \) This should be obvious to us, since \( S_1 \) accesses even elements of \( A \), and \( S_2 \) odd elements.

The GCD Test – Example II

\[
\begin{align*}
\text{FOR } I & := 1 \text{ TO } 10 \text{ DO} \\
S_1 & : \ A[19 \cdot I + 3] := \cdots \\
S_2 & : \cdots := A[2 \cdot I + 21] \\
\text{ENDFOR}
\end{align*}
\]

\( c \cdot I^d + j = d \cdot I^u + k \) only if gcd\((c, d)\) evenly divides \( k - j \), i.e. if \((k - j) \text{ mod gcd}(c, d) = 0 \).

\( c = 19, j = 3, d = 2, k = 21 \).

\( (21 - 3) \text{ mod gcd}(19, 2) = 18 \text{ mod } 2 = 0 \Rightarrow \) There’s a flow dependence: \( S_1 \longrightarrow S_2 \).

The only values that satisfy the dependence are \( I^d = 2 \) and \( I^u = 10: \ 19 \cdot 2 + 3 = 41 = 2 \cdot 10 + 21 \). If the loop had gone from 3 to 9, there would be no dependence! The gcd-test doesn’t catch this.

The GCD Test – Example III

\[
\begin{align*}
\text{FOR } I & := 1 \text{ TO } 10 \text{ DO} \\
S_1 & : \ A[8 \cdot i + 3] := \cdots \\
S_2 & : \cdots := A[2 \cdot i + 1] \\
\text{ENDFOR}
\end{align*}
\]

\( c \cdot I^d + j = d \cdot I^u + k \) only if gcd\((c, d)\) evenly divides \( k - j \), i.e. if \((k - j) \text{ mod gcd}(c, d) = 0 \).

\( c = 8, j = 3, d = 2, k = 1 \).

\( (1 - 3) \text{ mod gcd}(8, 2) = -2 \text{ mod } 2 = 0 \Rightarrow \) There’s a flow dependence: \( S_1 \longrightarrow S_2 \).

We knew this already, from the example in a previous slide. \( I^d = 1, I^u = 5 \Rightarrow 8 \cdot I^d + 3 = 11 = 2 \cdot I^u + 1 \).
Dependence Distance

FOR I := 2 TO 10 DO
    S1: A[I] := B[I] + C[I];
ENDFOR

- On each iteration, $S_1$ will assign a value to $A[i]$, and $S_2$ will use it.
- Therefore, there's a flow dependence from $S_1$ to $S_2$: $S_1 \delta S_2$.
- We say that the data-dependence direction for this dependence is $\geq$, since the dependence stays within one iteration.
- We write: $S_1 \delta \geq S_2$.

Dependence Directions II

FOR I := 2 TO 10 DO
    S1: A[I] := B[I] + C[I];
ENDFOR

- On each iteration, $S_1$ will assign a value to $A[i]$, and $S_2$ will use this value in the next iteration.
- E.g., in iteration 3, $S_1$ assigns a value to $A[3]$. This value is used by $S_2$ in iteration 4.
- Therefore, there's a flow dependence from $S_1$ to $S_2$: $S_1 \delta S_2$.
- We say that the data-dependence direction for this dependence is $<$, since the dependence flows from $i-1$ to $i$.
- We write: $S_1 \delta< S_2$.

Dependence Directions III

FOR I := 2 TO 10 DO
    S1: A[I] := B[I] + C[I];
ENDFOR

- On each iteration, $S_2$ will use a value that will be overwritten by $S_1$ in the next iteration.
- E.g., in iteration 3, $S_2$ uses the value in $A[4]$. This value is overwritten by $S_1$ in iteration 4.
- Therefore, there's a anti dependence from $S_2$ to $S_1$: $S_2 \overline{\delta} S_1$.
- We say that the data-dependence direction for this dependence is $<\overline{\delta}$, since the dependence flows from $i$ to $i+1$.
- We write: $S_2 \overline{\delta}< S_1$. 
Loop Nests

FOR \( I := 0 \) TO \( 9 \) DO
FOR \( J := 1 \) TO \( 10 \) DO
    \( S_1: \cdots := A[I, J-1] \)
    \( S_2: A[I, J] := \cdots \)
ENDFOR
ENDFOR

With nested loops the data-dependence directions become vectors. There is one element per loop in the nest.

In the loop above there is a flow dependence \( S_2 \rightarrow S_1 \) since the element being assigned by \( S_2 \) in iteration \( I \) \((A[I, J])\) will be used by \( S_1 \) in the next iteration.

This dependence is carried by the \( J \) loop.

We write: \( S_2 \delta_{=,<} S_1 \).

Loop Nests II – Example

FOR \( I := 1 \) TO \( N \) DO
FOR \( J := 2 \) TO \( N \) DO
    \( S_3: D[I, J] := 0.1 \);
ENDFOR
ENDFOR

\( S_1 \delta_{=,<} S_1 \) \( S_1 \) assigns a value to \( A[I, J] \) in iteration \((I, J)\) that will be used by \( S_1 \) in the next iteration \((I, J+1)\).

The dependence is carried by the \( J \) loop.

\( S_1 \delta_{=,=} S_2 \) \( S_1 \) assigns a value to \( A[I, J] \) in iteration \((I, J)\) that will be used by \( S_2 \) in the same iteration.

\( S_2 \delta_{<,=} S_3 \) \( S_2 \) uses the value of \( D[I + 1, J] \) in iteration \((I, J)\). It will be overwritten by \( S_3 \) in the next \( I \)-iteration. The \( I \)-loop carries the dependence.
Suppose we have the following loop-nest:

```
for i:=1 to x do
    for j := 1 to y do
        s1: A[a*i+b*j+c,d*i+e*j+f] = ... 
        s2: ... = A[g*i'+h*j'+k,l*i'+m*j'+n]
```

Then there is a dependency between statements $s_1$ and $s_2$ if there exist iterations $(i,j)$ and $(i',j')$, such that

$$a \cdot i + b \cdot j + c = g \cdot i' + h \cdot j' + k$$
$$d \cdot i + e \cdot j + f = l \cdot i' + m \cdot j' + n$$

or

$$a \cdot i - g \cdot i' + b \cdot j - h \cdot j' = k - c$$
$$d \cdot i - l \cdot i' + e \cdot j - m \cdot j' = n - f$$

This is equivalent to an integer programming problem (a system of linear equations with all integer variables and constants) in four variables:

$$\begin{bmatrix}
  a & -g & b & -h \\
  d & -l & e & -m
\end{bmatrix}
\begin{bmatrix}
  i \\
  i'
\end{bmatrix}
= 
\begin{bmatrix}
  k - c \\
  n - f
\end{bmatrix}$$

If the loop bounds are known we get some additional constraints:

$$1 \leq i \leq x, \quad 1 \leq i' \leq x,$$
$$1 \leq j \leq y, \quad 1 \leq j' \leq y$$

In other words, to solve this dependency problem we look for integers $i, i', j, j'$ such that the equation and constraints above are satisfied.

### Homework

1. What is the gcd-test? What do we mean when we say that the gcd-test is conservative?
2. List the data dependencies ($\rightarrow$, $\rightarrow+$, $\rightarrow-$) for the loops below.

```plaintext
FOR i := 1 TO 7 DO
    S1: ... := A[2*i + 1];
    S2: ... := A[4*i];
END;

FOR i := 1 TO n DO
    S2: A[2*i + 1] := X + B[i + 7];
    S3: A[i + 5] := C[10*i];
END;
```
Consider the following loop:

```plaintext
FOR i := 1 TO n DO
    S1: B[i] := C[i - 1] * 2;
    S3: D[i] := C[i] * 3;
    S4: C[i] := B[i - 1] + 5;
ENDFOR
```

List the data dependencies for the loop. For each dependence indicate whether it is a flow- (→), anti- (←→), or output-dependence (→◦), and whether it is a loop-carried dependence or not.

Show the data dependence graph for the loop.

### Readings and References


### Summary I

- Dependence analysis is an important part of any parallelizing compiler. In general, it’s a very difficult problem, but, fortunately, most programs have very simple index expressions that can be easily analyzed.
- Most compilers will try to do a good job on common loops, rather than a half-hearted job on all loops.
- Integer programming is NP-complete.
When faced with a loop

\[\text{FOR } i := \text{From TO To DO} \]
\[S_1: \ A[f(i)] := \cdots \]
\[S_2: \ \cdots := A[g(i)]\]

ENDFOR

the compiler will try to determine if there are any index values \(I, J\) for which \(f(I) = g(J)\). A number of cases can occur:

1. The compiler decides that \(f(i)\) and \(g(i)\) are too complicated to analyze. ⇒ Run the loop serially.
2. The compiler decides that \(f(i)\) and \(g(i)\) are very simple (e.g. \(f(i) = i, f(i) = c*i, f(i) = i+c, f(i) = c*i+d\)), and does the analysis using some built-in pattern matching rules. ⇒ Run the loop in parallel or serially, depending on the outcome.
3. The compiler applies some advanced method to determine the dependence. ⇒ Run the loop in parallel or serially, depending on the outcome.

Most compilers use pattern-matching techniques to look for important and common constructs, such as reductions (sums, products, min & max of vectors).

The simplest analysis of all is a name analysis: If every identifier in the loop occurs only once, there are no dependencies, and the loop can be trivially parallelized:

\[\text{FOR } i := \text{From TO To DO} \]
\[S_1: \ A[f(i)] := B[g(i)] + C[h(i)]; \]
\[S_2: \ D[j(i)] := E[k(i)] \times F[m(i)]; \]

ENDFOR