## Bernstein’s Conditions (1966)

- For a statement $S$:
  - $IN(S)$ = set of variables, registers, or locations used by $S$
  - $OUT(S)$ = set written to by $S$
- $S_1; S_2$ (sequence) is equivalent to $S_1 \parallel S_2$ (parallel) provided that:
  - $OUT(S_1) \cap OUT(S_2) = \emptyset$
  - $OUT(S_1) \cap IN(S_2) = \emptyset$
  - $OUT(S_2) \cap IN(S_1) = \emptyset$

## Data Dependence

- Expresses constraints on parallel execution, as derived from sequential execution semantics
- Types of Dependence (Kuck, Wolfe, et al.):
  - Flow dependence
  - Anti dependence
  - Output dependence

## Flow Dependence

- A variable assigned to in one statement is used in a later one:
  - $A = 5$
  - $B = A^2$

## Anti Dependence

- A variable used in one statement is assigned to in a later one:
  - $B = A^2$
  - $A = 5$
Output Dependence

- A variable assigned to in one statement is later re-assigned to:
  
  \[ A = B^2 \]
  
  \[ A = 5 \]  

Removable Dependences

- Anti Dependence and Output Dependence are removable.
- They are artifacts of using variables as if memory locations, rather than purely for their values.
- Flow Dependence is not removable (unless the algorithm is changed); it expresses essential precedence.
- Clarification of whether location- or value-based dependency is being considered will be left to context.

Notation

- \( S_1 \delta S_2 \) means \( S_2 \) is flow dependent on \( S_1 \)
- \( S_1 \delta^a S_2 \) means \( S_2 \) is anti dependent on \( S_1 \)
- \( S_1 \delta^o S_2 \) means \( S_2 \) is output dependent on \( S_1 \)

Dependence Relations determine a Partial Order on Statement Execution

- Statements must be done in sequence
- Statements can be done in either order, or in parallel

Location- vs. Value-Based

- Consider
  
  \[ A = 5 \]  
  
  \[ B = A + 7 \]  
  
  \[ A = 99 \]  
  
  \[ C = A^2 \]  

By using a different variable, the dependency is removed

- Consider
  
  \[ A = 5 \]  
  
  \[ B = A + 7 \]  
  
  \[ AA = 99 \]  
  
  \[ C = AA^2 \]
### Loops add to the Challenge

- Consider for \( K = 1 \) to \( 10 \)
  - \( S_i(K) \quad A[K] = B[K] \)

- Conclude: All instances \( S_i(K) \) can be done **concurrently** (since no arrows).

### Larger offsets allow more concurrency

- Consider for \( K = 3 \) to \( 10 \)

- \( S_i(3) \parallel S_i(4) \) is possible
- \( S_i(K) \parallel S_i(K+1) \) is possible, \( K = 3, 5, \ldots \)

### Transformation reduces sequence constraints

- **F90 style:**
  
  \[
  \begin{align*}
  B(1 : 9) &= A(2 : 10) \\
  A(2 : 10) &= B(2 : 10)
  \end{align*}
  \]
The type of transformation just shown can be automated. This is done routinely in compilers for high-performance machines.

**Parallel Execution of Loops**

**Strategy**
- Try to issue different instances of a *loop body* to separate processing elements.
- Generally loops occur nested; try to find appropriate nesting level where different instances can be issued.

**Similar issue to Parallelization: Vectorization**
- Vector machines:
  - Exploit fine-grain parallel operations (+, *, /) on *vector* elements
  - Typically done with *vector* registers
- Vectorizing concentrates on *inner* loop
- Parallelizing concentrates on *outer* loops (coarser grain)

**Example of Loop Vectorization**

|---------------|-------------------|------------|

Vectorizes to (using F90 notation):
- \( A(1:N) = B(1:N) + C(1:N) \)
- \( D(1:N) = A(1:N)*5 \)

**Example of Loop Vectorization**

<table>
<thead>
<tr>
<th>for K = N to 1 by -1</th>
<th>A[K] = ... ( = A[K + d] )</th>
<th>( d &gt; 0 \Rightarrow \text{old value used} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>... = A[K - d] ( d &lt; 0 \Rightarrow \text{new value used} )</td>
<td>since assignment order is ( A[1], A[2], ... ) but use order is ( A[1+d], A[2+d], ... )</td>
</tr>
</tbody>
</table>

**Mnemonic Aids**

<table>
<thead>
<tr>
<th>for K = 1 to N</th>
<th>A[K] = ... ( = A[K + d] )</th>
<th>( d &gt; 0 \Rightarrow \text{old value used} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>... = A[K - d] ( d &lt; 0 \Rightarrow \text{new value used} )</td>
<td>since assignment order is ( A[N], A[N-1], ... ) but use order is ( A[N-d], A[N-1-d], ... )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>for K = 1 to N</th>
<th>A[K] = ... ( = A[K + d] )</th>
<th>( d &gt; 0 \Rightarrow \text{old value used} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>... = A[K - d] ( d &lt; 0 \Rightarrow \text{new value used} )</td>
<td>since assignment order is ( A[1], A[2], ... ) but use order is ( A[1+d], A[2+d], ... )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
A&(K) = B(K) + C(K) \\
D&(K) = A(K+1)*5
\end{align*}
\]

\[
\begin{align*}
A&(1:N) = B(1:N) + C(1:N) \\
D&(1:N) = A(1:N)*5
\end{align*}
\]
Dependence Distance

Extending our notation of statement dependence:

\[ S_0 \delta_x^{(d)} S_1 \]

where \( x \in \{f, a, o\} \)

\[ \delta_x \] is a signed integer

Says:

- For each \( i \), \( S_0(i) \) must be done before \( S_1(i + d) \).

For \( K = 2 \) to \( N-1 \),

\[ S_1 C[K] = A[K-1] \]

For each \( i \), \( S_0(i) \) must be done before \( S_1(i + 1) \).

Both Indices and Loop Direction must be taken into account in determining Dependence Distance.

For \( K = 2 \) to \( N-1 \),

\[ S_1 C[K] = A[K-1] \]

is similar to

- For \( K = N-1 \) to \( 2 \) by \(-1\),

\[ S_1 C[K] = A[K+1] \]

in that \( C[K] \) gets the new value, not the old.

In general, there may be a different set of dependence distances for each array:

For \( K = 2 \) to \( N \),

\[ S_1 C[K] = A[K] \]

Each places a constraint on loop restructuring.

Example

- For \( K = 2 \) to \( N \),

\[ S_1 C[K] = A[K-1] \]

or

\[ \delta_{(1)} \] or \( \delta_{(<)} \)

- For \( K = 1 \) to \( N-1 \),

\[ S_1 C[K] = A[K+1] \]

or

\[ \delta_{(1)} \] or \( \delta_{(<)} \)

Direction Vectors

- Less precise than Dependence Distances, but frequently used:

  - \( \delta_{(a)} \) used in place of \( \delta_{(d)} \) where \( d > 0 \)
  - \( \delta_{(a)} \) used in place of \( \delta_{(0)} \)
  - \( \delta_{(a)} \) used in place of \( \delta_{(d)} \) where \( d < 0 \)

  - An advantage of using \( > \) is that \( d \) might not be fixed, as in:

    - For \( K = 2 \) to \( 10 \),

    - Here the dependence distance increases with \( K \).
**doacross (M. Wolfe)**

- doacross K = M to N

  is equivalent to HPF’s INDEPENDENT annotation:

  Each loop body is done independently of the others, possibly in parallel (There is still sequencing **within** the body.)

**doacross Example**

- Original loop

  for K = 1 to N
  
  \[ A[K] = C[K] \]
  

- is optimized to

  doacross K = 1 to N
  
  \[ A[K] = C[K] \]
  

**Non-doacross Example**

- Original loop

  for K = 2 to N
  
  \[ A[K] = C[K] \]
  

  cannot be optimized using doacross alone.

- We could provide additional synchronization on the use of A[K-1] to do it, but it wouldn’t be pure doacross.

**Loops that “Carry” Dependence**

- As we saw, loops having only \( \Delta_f = \) are optimizable using doacross.

- A loop with \( \Delta_f < \) or \( \Delta_f > \) carries a dependence that prevents parallel execution.

**Nested Loops**

- For nested loops, a vector of dependences is used, e.g. \( \Delta_f, \Delta_a = \) with one component per loop nest.

- When loops are nested, the **outermost** loop with a \( \Delta_f < \) or \( \Delta_f > \) carries the dependence.

**Nested Loop Example**

- for K = 2 to N

  for J = 2 to N
  
  

- The inner loop carries a dependence for A; no loop carries a dependence for B.

- Therefore the outer loop can be parallelized using doacross.
## Nested Loop Example

- for K = 2 to N
  - for J = 2 to N

- doacross K = 2 to N
  - for J = 2 to N

## Exercise

- How to parallelize:
  - for K = 2 to N
    - for J = 2 to N

## Loop Interchanging

- for K = 1 to N
  - for J = 2 to N

- \( S_1 \) \( \delta'_{(\leq, \leq)} \) \( S_1 \) implies inner loop cannot be vectorized.
- No dependencies of form \( \delta'_{(\leq, >)} \) implies loops can be interchanged

## Exercise

- for K = 2 to N
  - for J = 2 to N

- Parallel:
  - doacross J = 2 to N

- for K = 1 to N
  - for J = 2 to N

- for J = 2 to N
  - for K = 1 to N

- Now have \( \delta'_{(\leq, \leq)} \)
Loop Interchanging

- for J = 2 to N
  - for K = 1 to N
- Now have δ[<, <]
- Execute as:
  - for J = 2 to N

Parallelization Rules
(as summarized by Thomas Bräunl)

- Data dependences ‘=’ for the target loop do not have to be synchronized.
- Data dependences ‘<’ or ‘>’ in loops outside the target do not have to be considered.
- Loops inside the target do not have to be considered.
- All other data dependences need to be synchronized, such as with semaphores.
- Execution order inside the target loop may be changed, but ‘=’ dependences must translate into enforced precedences.

Rule Examples

- Data dependence directions with = in the target loop do not have to be synchronized.
  - for i = 1 to n
    - S_1: A[i] = C[i]
    - S_2: B[i] = A[i]
    - doacross i = 1 to n
    - S_1: A[i] = C[i]
    - S_2: B[i] = A[i]
    - S_1 δ[=, =] S_2

Rule Examples

- Data dependences ‘<’ or ‘>’ in loops outside the target do not have to be considered.
  - for i = 1 to n
    - for j = 1 to m
      - S_1: A[i, j] = C[i, j]
      - S_2: B[i, j] = A[i-1, j-1]
      - doacross j = 1 to m
      - S_1: A[i, j] = C[i, j]
      - S_2: B[i, j] = A[i-1, j-1]
      - S_1 δ[<, <] S_2

Rule Examples

- Loops inside the target do not have to be considered.
  - for i = 1 to n
    - for j = 1 to m
      - S_1: A[i, j] = B[i, j]
      - doacross j = 1 to m
      - S_1: A[i, j] = B[i, j]
      - S_1 δ[=, <] S_2

Gauss-Seidel & Wavefronts

- Outline
  - Gauss-Seidel vs. Jacobi
  - Wavefronts
  - Red-black
  - Chaotic approaches