Review: Affine Loop Analysis

• A function $f$ of variables $x_1, x_2, ..., x_n$ is affine
  - If it is in such a form
    • $f = c_0 + c_1 x_1 + c_2 x_2 + ... + c_n x_n$, where $c_i$ are all constants
    • also called a linear function

• Affine array accesses
  - The indexes of an array element are all affine functions of loop indexes
  - E.g., $z[i][i + 2*j - I]$

• Affine loops
  - Each loop has a single loop induction variable
  - The bounds of each loop are affine expressions of outer loop induction variables
Review: Affine Transformation Theory

• **Affine spaces**
  - **Iteration Space**
    • the set of dynamic execution instances
    • i.e. the set of value vectors taken by loop indices
      • a \( k \)-dimensional space for a \( k \)-level loop nest
  - **Data Space**
    • the set of array elements accessed
    • an \( n \)-dimensional space for an \( n \)-dimensional array
  - **Processor Space**
    • the set of processors in the system
    • in analysis, we may pretend there are unbounded \# of virtual processors

```c
float Z[100];
for (i=0; i<10; i++)
    Z[i+10] = Z[i];
```
Review: Data Space

• An Example

float Z[100];
for (i=0; i<10; i++)
    Z[i+10] = Z[i];
f_R(i) = i
f_W(i) = i+10
Review: Iteration Spaces

• Assumptions
  - Each loop has a single loop index.
  - It increments by 1 at each iteration.
  - The bounds of each loop are affine expressions of outer loop indices.

```c
for (i = 0; i <= 5; i++)
  for (j = i; j <= 7; j++)
    Z[j, i] = 0;
```
Review: Data Space

- Each array index is expressed as affine expressions of loop induction variables and symbolic constants.
- A loop is affine if
  1. The loop bounds are affine expressions.
  2. Array indexes are affine expressions.
- \(< F, f, B, b >\) representation
  - Maps a vector \(i\) within \(B*i + b > 0\) to array element location \(F*i + f\).
  - \(B\) and \(b\) are for loop bounds
  - \(F\) and \(f\) are for memory references: \(F\) is the corresponding d-column matrix and \(f\) is the d-row vector. (d for the # of loop levels)

F: coefficient matrix
Review: Data Space

- \(< \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} >\) representation
  - Maps a vector \(i\) within \(\mathbf{B}i + \mathbf{b} > 0\) to array element location \(\mathbf{F}i + \mathbf{f}\).
  - \(\mathbf{B}\) and \(\mathbf{b}\) are for loop bounds
  - \(\mathbf{F}\) and \(\mathbf{f}\) are for memory references: \(\mathbf{F}\) is the corresponding \(d\)-column matrix and \(\mathbf{f}\) is the \(d\)-row vectors. (\(d\) for the \# of loop levels)

\[
\begin{align*}
\mathbf{B} + \mathbf{b} &= 0: \\
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 0 \\
-1 & 1 & 0 \\
0 & -1 & 0 \\
0 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
k
\end{bmatrix}
&= \\
\begin{bmatrix}
0 & 1 & 0 \\
5 & 0 & 0 \\
0 & 0 & 0 \\
7 & 0 & 0 \\
9 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}

\mathbf{F} + \mathbf{f}:
\begin{align*}
\begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
k
\end{bmatrix}
&= \\
\begin{bmatrix}
0 \\
1 \\
0 \\
4
\end{bmatrix}
\end{align*}
\]

for (\(i=0; i<=5; i++\))
for (\(j=i; j<=7; j++\))
for (\(k=j; k<=9; k++\))
\(Z[j, i+1, 4] = 0;\)
Review: Processor Space

- \(<C, c>\) to represent how to map iterations to processor (at space)
  1. \(C\) is a \(n \times m\) matrix
     - \(m=d\) (the loop level)
     - \(n\) is the dimension of the processor grid
  2. \(c\) is a \(n\)-element constant vector
  3. \(p = C*i + c\)

- Examples

  **1-d processor grid**
  
  ```cpp
  for (i=1; i<=N; i++)
  Y[i] = Z[i];
  ```

  \(C = [1], c = [0], p = i\)

  **2-d processor grid**
  
  ```cpp
  for (i=1; i<=N; i++)
  for (j=1; j<=N; j++)
  Y[i,j] = Z[i,j];
  ```

  \(C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, c = [0\ 0], p = i, q = j\)

**Notation:**
- **bold fonts** for container variables;
- **normal fonts** for scalar variables.
Review: Data Dependence

• Definition
  Given two memory references, there exists a dependence between them if the three following conditions hold:
  ✓ They reference the same array (cell)
  ✓ One of them is a write
  ✓ The two associated statements are executed

• Two memory accesses \(<F, f, B, b>\) and \(<F', f', B', b'>\) are data dependent iff
  - At least one of them is a write reference and
  - There exist two vectors \(i\) and \(i'\) such that
    \[
    \begin{align*}
    B*i + b & \geq 0 \\
    B'i + b' & \geq 0 \\
    F*i + f & = F'*i' + f'.
    \end{align*}
    \]
    Where \(i \neq i'\).

Note for instances of the same access, we need to add constraint \(i \neq i'\).
Loop Parallelization

• Where there is no dependence between loop iterations
  - Maximum amount of parallelism
  - How to check if there is dependence between loop iterations?

• Where there is dependence between loop iterations
  - Synchronization Free
  - Divide loop iterations into multiple partitions
  - Each partition is mapped to one processor
  - Dependence may only exist within one partition
  - Synchronize Needed
  - Divide loop iterations into multiple partitions
  - Each partition is mapped to one time tick
  - Dependence does not exist within a partition
Review: Integer Linear Programming (ILP)

• Existence of data dependence means existence of integer solutions for
  1. \( B^*i + b \geq 0 \)
  2. \( B'^*i' + b' \geq 0 \)
  3. \( F^*i + f = F'^*i' + f' \) is essentially the following:
    \[
    \begin{align*}
    F^*i + f &\leq F'^*i' + f' \\
    F^*i + f' &\geq F'^*i' + f'
    \end{align*}
    \]

• Integer linear programming
  - Problem to solve:
    Finding integer solutions for a set of linear inequalities.
  - Complexity: NP-complete.
  - Heuristic approaches exist.
Solving ILP for Data Dependence

• Does data dependence exist in a loop?
  - Checking if there is solution to the integer linear programming problem
  - Equivalent to checking if a polyhedron is empty

• Three steps
  1. GCD test (use Diophantine Equation)
     - If failed, no data dependences, otherwise, continue.
  2. Use a set of heuristics to examine the inequalities (loop residue graph and etc)
     - If still not sure, continue.
  3. Branch-and-bound approach (a general ILP approach)

\[
\begin{align*}
1. \ B^*i + b & \geq 0 \\
2. \ B'^*i' + b' & \geq 0 \\
3. \ F^*i + f & = F'^*i' + f' \\
    (a). \ F^*i + f & \leq F'^*i' + f' \\
    (b). \ F^*i + f & \geq F'^*i' + f'
\end{align*}
\]
Step 3

• Branch and Bound
  - A general method for integer linear programming problems
  - Split the space into half whenever search fails

A. Generate loop bounds:

\[ S_n = S; \]
for (i=n; i>=1; i--){
  \[ L_{vi} = \text{all the lower bounds on } v_i \text{ in } S_i; \]
  \[ U_{vi} = \text{all the upper bounds on } v_i \text{ in } S_i; \]
  \[ S_{i-1} = \text{Constraints by eliminating } v_i \text{ from } S_i; \]
*/ remove redundancies */
\[ S' = \Phi; \]
for (i=1; i<=n; i++){
  Remove any bounds in \( L_{vi} \) and \( U_{vi} \) implied by \( S' \);
  Add the remaining constraints of \( L_{vi} \) and \( U_{vi} \) on \( v_i \) to \( S' \);
}

B. Check if an integer solution exists

Apply Algorithm A to \( S_n \) to project away variables \( v_n, v_{n-1}, \ldots, v_1 \) in that order
Let \( S_i \) be the polyhedron after projecting away \( v_{i+1} \), for
\( i = n-1, n-2, \ldots, 0; \)
if \( S_0 \) is false return “no”;
for (i = 1; i <= n; i++)
{
  if (\( S_i \) does not include an integer value) break;
  pick \( c_i \), an integer in the middle of the range for \( v_i \) in \( S_i \);
  modify \( S_i \) by replacing \( v_i \) by \( c_i \);
}
if ( \( i == n+1 \) ) return “yes”;
if ( \( i == 1 \) ) return “no”;
let the lower and upper bounds on \( v_i \) in \( S_i \) be
\( l_i \) and \( v_i \) respectively;
recursively apply this algorithm to \( S_n \cup \{ v_i <= \text{floor}(l_i) \} \) and
\( S_n \cup \{ v_i >= \text{ceiling}(u_i) \} \);
if (either returns “yes”) return “yes” else return “no”;

Chapter 11 of “Compilers principles, techniques, and tools” (2nd Ed) by Aho, Lam, Sethi, and Ullman.
B. Check if an integer solution exists

Apply Algorithm A to $S_n$ to project away variables $v_n, v_{n-1}, \ldots, v_1$ in that order

Let $S_i$ be the polyhedron after projecting away $v_{i+1}$, for $i = n-1, n-2, \ldots, 0$;

if $S_0$ is false return “no”;

for $(i = 1; i <= n-1; i++)$
{
    if ($S_{i-1}$ does not include an integer value) break;
    pick $c_i$, an integer in the middle of the range for $v_i$ in $S_i$;
    modify $S_i$ to $S_n$ by replacing $v_i$ by $c_i$;
}

if (i == n+1) return “yes”;

if (i == 1) return “no”;

let the lower and upper bounds on $v_i$ in $S_i$ be $l_i$ and $v_i$ respectively;

recursively apply this algorithm to $S_n \cup \{v_i \leq \text{floor}(l_i)\}$ and $S_n \cup \{v_i \geq \text{ceiling}(u_i)\}$;

if (either returns “yes”) return “yes” else return “no”;

---

**Example 1:**

- Let’s check the bounds for $x$
- The integer point in the middle of the range $[0, 2.7]$ is 1.
- Let $x = 1$. 

Iteration 1:

- $(y, x)$
- $x: [0, 2.7]$
Branch and Bound

• Example 1:

B. Check if an integer solution exists
Apply Algorithm A to $S_n$ to project away variables $v_n, v_{n-1}, \ldots, v_1$ in that order
Let $S_i$ be the polyhedron after projecting away $v_{i+1}$, for $i = n-1, n-2, \ldots, 0$;
if $S_0$ is false return “no”;
for $(i = 1; i <= n; i++)$
{
    if ($S_i$ does not include an integer value) break;
    pick $c_i$, an integer in the middle of the range for $v_i$ in $S_i$;
    modify $S_i$ by replacing $v_i$ by $c_i$;
}
if ($i == n+1$) return “yes”;
if ($i == 1$) return “no”;
let the lower and upper bounds on $v_i$ in $S_i$ be $l_i$ and $v_i$ respectively;
recursively apply this algorithm to $S_n U \{v_i <= floor(l_i)\}$ and $S_n U \{v_i >= ceiling[u_i]\}$;
if (either returns “yes”) return “yes” else return “no”;
**Branch and Bound**

**Example 1:**

- **x** = 1.5
- **y** ∈ [0.05, 1.3]

Not an integer!

**Iteration 2:**

- Pick an integer in the middle of the range of **y** = [0.05, 1.3]
- Let **y** = 1 (x = 1). Integer point found!
- This polyhedron has an integer solution.

**B. Check if an integer solution exists**

Apply Algorithm A to **S**\(_n\) to project away variables **v**\(_n\), **v**\(_{n-1}\), ..., **v**\(_1\) in that order

Let **S**\(_i\) be the polyhedron after projecting away **v**\(_{i+1}\), for i = n-1, n-2, ..., 0;

if **S**\(_0\) is false return “no”;

for (i = 1; i <= n; i++)
{
    if (**S**\(_i\) does not include an integer value) break;
    pick **c**\(_i\), an integer in the middle of the range for **v**\(_i\) in **S**\(_i\);
    modify **S**\(_i\) - **S**\(_n\) by replacing **v**\(_i\) by **c**\(_i\);
}

if (i == n+1) return “yes”;
if (i == 1) return “no”;

let the lower and upper bounds on **v**\(_i\) in **S**\(_i\) be **l**\(_i\) and **v**\(_i\) respectively;

recursively apply this algorithm to **S**\(_n\) U {**v**\(_i\) <= floor(**l**\(_i\))} and
**S**\(_n\) U {**v**\(_i\) >= ceiling[**u**\(_i\)]};

if (either returns “yes”) return “yes” else return “no”;

**Example 1:**

- **x** = 1.5
- **y** ∈ [0.05, 1.3]

Not an integer!
Branch and Bound

• Example 2:

B. Check if an integer solution exists

Apply Algorithm A to $S_n$ to project away variables $v_n, v_{n-1}, \ldots, v_1$ in that order.

Let $S_i$ be the polyhedron after projecting away $v_{i+1}$, for $i = n-1, n-2, \ldots, 0$;

if $S_0$ is false return “no”;

for ($i = 1; i \leq n; i++$)

{ if ($S_i$ does not include an integer value) break;

    pick $c_i$, an integer in the middle of the range for $v_i$ in $S_i$;
    modify $S_i$ by replacing $v_i$ by $c_i$;

}

if ($i == n+1$) return “yes”;

if ($i == 1$) return “no”;

let the lower and upper bounds on $v_i$ in $S_i$ be $l_i$ and $v_i$ respectively;

recursively apply this algorithm to $S_n \cup \{v_i \leq \text{floor}(l_i)\}$ and $S_n \cup \{v_i \geq \text{ceiling}[u_i]\}$;

if (either returns “yes”) return “yes” else return “no”;

Iteration 1:

• Let’s check the bounds for $x$
• The integer point in the middle of the range $[-0.1, 2.4]$ is $x = 1$. 
**Branch and Bound**

- **Example 2:**

![Diagram showing a polyhedron with no integer point]

B. Check if an integer solution exists

Apply Algorithm A to \( S_n \) to project away variables \( v_n, v_{n-1}, \ldots, v_1 \) in that order.

Let \( S_i \) be the polyhedron after projecting away \( v_{i+1} \), for \( i = n-1, n-2, \ldots, 0; \)

if \( S_0 \) is false return “no”;

for (\( i = 1; i <= n; i++ \))

\{
    if (\( S_i \) does not include an integer value) break;
    pick \( c_i \), an integer in the middle of the range for \( v_i \) in \( S_i \);  
    modify \( S_i \) by replacing \( v_i \) by \( c_i \);
\}

if (\( i == n+1 \)) return “yes”;

if (\( i == 1 \)) return “no”;

let the lower and upper bounds on \( v_i \) in \( S_i \) be \( l_i \) and \( u_i \) respectively;

recursively apply this algorithm to \( S_n \cup \{ v_i <= floor(l_i) \} \) and \( S_n \cup \{ v_i >= ceiling(u_i) \} \);

if (either returns “yes”) return “yes” else return “no”;

Iteration 1 (cont.):

- Modify \( S \) by replacing \( x \) by 1.
- The modified polyhedron becomes a line segment: \( x = 1, y = [0.05, 0.95] \)


Branch and Bound

• Example 2:

B. Check if an integer solution exists
Apply Algorithm A to $S_n$ to project away variables $v_n, v_{n-1}, \ldots, v_1$ in that order
Let $S_i$ be the polyhedron after projecting away $v_{i+1}$, for $i = n-1, n-2, \ldots, 0$;
if $S_0$ is false return “no”;
for ($i = 1; i <= n; i++$)
{
    if ($S_i$ does not include an integer value) break;
    pick $c_i$, an integer in the middle of the range for $v_i$ in $S_i$;
    modify $S_i$ by replacing $v_i$ by $c_i$;
}
if (i == n+1) return “yes”;
if (i == 1) return “no”;
let the lower and upper bounds on $v_i$ in $S_i$ be
$l_i$ and $v_i$ respectively;
recursively apply this algorithm to $S_n \cup \{v_i <= \text{floor}(l_i)\}$ and
$S_n \cup \{v_i >= \text{ceiling}(u_i)\}$;
if (either returns “yes”) return “yes” else return “no”;

Branch and bound
• Two branches (two polyhedrons):
  $S_1 \cup \{y >= 1\}, S_1 \cup \{y <= 0\}$
• Search for integer solution for both branches, one at a time.
Loop Parallelization

• Where there is no dependence between loop iterations
  - Maximum amount of parallelism
  - How to check if there is dependence between loop iterations

• Where there is dependence between loop iterations
  - Divide loop iterations into multiple partitions
  - Synchronization Free
    • Each partition → one processor
    • Dependence only exists within one partition
  - Synchronize Necessary
    • Each partition → one time tick
    • Dependence only exists across partition
Loop Parallelization

• Where there is no dependence between loop iterations
  - Maximum amount of parallelism
  - How to check if there is dependence between loop iterations

• Where there is dependence between loop iterations
  - Divide loop iterations into multiple partitions
  - Synchronization Free
    • Each partition → one processor
    • Dependence only exists within one partition
  - Synchronize Necessary
    • Each partition → one time tick
    • Dependence only exists across partition
Affine Parallel Schedule

forall (p=1; p<=N; p++){
    Y[p] = Z[p];
    X[p] = Y[p];
}  

• Affine Transformation for Parallelization
  - Consider outermost loop as parallel schedule
  - Each iteration of the outermost loop is running on an independent processor
Affine Parallel Schedule

• Affine Transformation for Parallelization
  - Consider outermost loop as parallel schedule
  - Each iteration of the outermost loop is running on an independent processor

```c
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        X[i,j] = X[i,j] + Y[i-1, j];  /* S1 */
        Y[i,j] = Y[i,j] + X[i, j-1];  /* S2 */
    }
```
Affine Transformation for Processor Schedule

- \(<C, c>\) to represent a partition
  1. \(C\) is a \(n \times m\) matrix
    - \(m = d\) (the loop level)
    - \(n\) is the dimension of the processor grid
  2. \(c\) is a \(n\)-element constant vector
  3. \(p = Ci + c\)
- Examples
  
  **1-d processor grid**
  
  for (i=1; i<=N; i++)
  
  \(Y[i] = Z[i];\)

  \(C = [1], c = [0], p = i\)

  **2-d processor grid**

  for (i=1; i<=N; i++)
  
  for (j=1; j<=N; j++)
  
  \(Y[i,j] = Z[i,j];\)

  \(C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, c = [0 0], p = i, q = j\)

**Notation:**
- **bold fonts** for container variables;
- **normal fonts** for scalar variables.
Synchronization-Free Constraints

- Two static accesses as \(<F_1, f_1, B_1, b_1>\) and \(<F_2, f_2, B_2, b_2>\) respectively in \(d_1\)-deep and \(d_2\)-deep loops
- Let \(<C_1, c_1>\) and \(<C_2, c_2>\) represent their respective partition functions
- To be synch-free
  - For all \(i_1\) in \(\mathbb{Z}_{d_1}\) (\(d_1\)-dimension integer vectors) and \(i_2\) in \(\mathbb{Z}_{d_2}\) such that
    1. \(B_1*i_1 + b_1 >= 0\), and
    2. \(B_2*i_2 + b_2 >= 0\), and
    3. \(F_1*i_1 + f_1 = F_2*i_2 + f_2\),

  it must be the case that \(C_1*i_1 + c_1 = C_2*i_2 + c_2\).
Example

- Goal: to find the processor schedule for each of the two statements. Denoted as

\[
p(S1): < [C_{11}, C_{12}], [c_1]> \\
p(S2): < [C_{21}, C_{22}], [c_2]> \]

assume 1-d processor grid
Step 1: Create Constraints

Consider dependence between $X[i, j]$ & $X[i, j-1]$:

for (i=1; i<=100; i++)
for (j=1; j<=100; j++){
    $X[i, j] = X[i, j] + Y[i-1, j]$;    /* S1 */
    $Y[i, j] = Y[i, j] + X[i, j-1]$;    /* S2 */
}

Linear constraints for accessing the same memory location

\[
\begin{align*}
(i_1, j_1): & \ 1 \leq i_1 \leq 100, \\
& \ 1 \leq j_1 \leq 100
\end{align*}
\]

\[
\begin{align*}
(i_2, j_2): & \ 1 \leq i_2 \leq 100, \\
& \ 1 \leq j_2 \leq 100
\end{align*}
\]
Step 1: Create Constraints

Consider dependence between $X[i,j]$ & $X[i,j-1]$: 

for (i=1; i<=100; i++)
for (j=1; j<=100; j++)
    
    $X[i, j] = X[i, j] + Y[i-1, j];$ /* S1 */

    $Y[i, j] = Y[i, j] + X[i, j-1];$ /* S2 */

Linear constraints for synchronization free parallelism

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} c_1 \end{bmatrix} = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} c_2 \end{bmatrix}$$

For $(i_1, j_1): 1 \leq i_1 \leq 100, 1 \leq j_1 \leq 100$

For $(i_2, j_2): 1 \leq i_2 \leq 100, 1 \leq j_2 \leq 100$
Step 1: Create Constraints

Consider dependence between $Y[i,j]$ & $Y[i-1,j]$:

for (i=1; i<=100; i++)
for (j=1; j<=100; j++){
    $X[i, j] = X[i, j] + Y[i-1, j]$; /* S1 */
    $Y[i, j] = Y[i, j] + X[i, j-1]$; /* S2 */
}

Step 1: Create Constraints

$(i_4, j_4): 1 \leq i_4 \leq 100$
$1 \leq j_4 \leq 100$

Linear constraints for accessing the same memory location

\[
\begin{cases}
    i_3 = i_4 - 1 \\
    j_3 = j_4
\end{cases}
\]
Step 1: Create Constraints

Consider dependence between $Y[i,j]$ & $Y[i-1,j]$:

for (i=1; i<=100; i++)
for (j=1; j<=100; j++){
    $X[i, j] = X[i, j] + Y[i-1, j]$; /* S1 */
    $Y[i, j] = Y[i, j] + X[i, j-1]$; /* S2 */
}

```
for (i=1; i<=100; i++)
for (j=1; j<=100; j++){
    X[i, j] = X[i, j] + Y[i-1, j]; /* S1 */
    Y[i, j] = Y[i, j] + X[i, j-1]; /* S2 */
}
```

Step 1: Create Constraints

(i, j): 1 <= i <= 100
       1 <= j <= 100

Linear constraints for synchronization free parallelism

$$[C_{11} \ C_{12}] \begin{bmatrix} i_3 \\ j_3 \end{bmatrix} + [c_1] = [C_{21} \ C_{22}] \begin{bmatrix} i_4 \\ j_4 \end{bmatrix} + [c_2]$$
Step 2 Reduce Unknowns

Apply Gaussian Elimination to

$$\mathbf{F}_1 \cdot \mathbf{i}_1 + f_1 = \mathbf{F}_2 \cdot \mathbf{i}_2 + f_2$$

1 <= \( i_1 \) <= 100, 1 <= \( j_1 \) <= 100,
1 <= \( i_2 \) <= 100, 1 <= \( j_2 \) <= 100,
\( i_1 = i_2 \), \( j_1 = j_2 - 1 \),

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{12} & C_{22}
\end{bmatrix}
\begin{bmatrix}
i_1 \\
j_1
\end{bmatrix}
+ [c_1] =
\begin{bmatrix}
C_{21} & C_{22} \\
C_{22} & C_{22}
\end{bmatrix}
\begin{bmatrix}
i_2 \\
j_2
\end{bmatrix}
+ [c_2]
\]

\[
\begin{bmatrix}
C_{11} - C_{21} & C_{12} - C_{22}
\end{bmatrix}
\begin{bmatrix}
i_3 \\
j_3
\end{bmatrix}
+ [c_1 - c_2 - C_{22}]
= 0
\]

1 <= \( i_3 \) <= 100, 1 <= \( j_3 \) <= 100,
1 <= \( i_4 \) <= 100, 1 <= \( j_4 \) <= 100,
\( i_3 = i_4 \), \( j_3 = j_4 \),

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{12} & C_{22}
\end{bmatrix}
\begin{bmatrix}
i_3 \\
j_3
\end{bmatrix}
+ [c_1] =
\begin{bmatrix}
C_{21} & C_{22} \\
C_{22} & C_{22}
\end{bmatrix}
\begin{bmatrix}
i_4 \\
j_4
\end{bmatrix}
+ [c_2]
\]

\[
\begin{bmatrix}
C_{11} - C_{21} & C_{12} - C_{22}
\end{bmatrix}
\begin{bmatrix}
i_3 \\
j_3
\end{bmatrix}
+ [c_1 - c_2 + C_{21}]
= 0
\]
Step 3 Solving the Equations

\[
\begin{bmatrix}
C_{11} - C_{21} & C_{12} - C_{22} \\
\end{bmatrix}
\begin{bmatrix}
i_1 \\
\end{bmatrix}
+ \begin{bmatrix}
c_1 - c_2 - C_{22} \\
\end{bmatrix} = 0
\Rightarrow C_{11} - C_{21} = 0, \ C_{12} - C_{22} = 0, \ \& \ c_1 - c_2 - C_{22} = 0.
\]

\[
\begin{bmatrix}
C_{11} - C_{21} & C_{12} - C_{22} \\
\end{bmatrix}
\begin{bmatrix}
i_3 \\
\end{bmatrix}
+ \begin{bmatrix}
c_1 - c_2 + C_{21} \\
\end{bmatrix} = 0
\Rightarrow C_{11} - C_{21} = 0, \ C_{12} - C_{22} = 0, \ \& \ c_1 - c_2 + C_{21} = 0.
\]

\[
C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1
\]
Solutions

Affine schedule for S1: \[
\begin{bmatrix}
C_{11} & C_{12}
\end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad c_1 = -1
\]
i.e. (i,j) iteration of S1 to processor \( p = i-j-1 \);

Affine schedule for S2: \[
\begin{bmatrix}
C_{21} & C_{22}
\end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad c_2 = 0
\]
i.e. (i,j) iteration of S2 to processor \( p = i-j \).

for (i=1; i<=100; i++)
for (j=1; j<=100; j++){
    X[i,j] = X[i,j] + Y[i-1, j]; /* S1 */
    Y[i,j] = Y[i,j] + X[i, j-1]; /* S2 */
}

\[ C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1 \]
Affine schedule for S1: $\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}$, $c_1 = -1$

i.e. $(i,j)$ iteration of S1 to processor $p = i-j-1$;

Affine schedule for S2: $\begin{bmatrix} C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}$, $c_2 = 0$

i.e. $(i,j)$ iteration of S2 to processor $p = i-j$. 

$p(S1): < [C_{11} \ C_{12}], [c_1]>$

$p(S2): < [C_{21} \ C_{22}], [c_2]>$
Affine Parallel Schedule

for (i=1; i<=100; i++)
  for (j=1; j<=100; j++){
    X[i,j] = X[i,j] + Y[i-1, j]; /* S1 */
    Y[i,j] = Y[i,j] + X[i, j-1]; /* S2 */
  }

• Affine Transformation for Parallelization
  - Consider outermost loop as parallel schedule
  - Each iteration of the outermost loop is running on an independent processor
Affine Parallel Schedule

- Affine Transformation for Parallelization
  - Consider outermost loop as parallel schedule
  - Each iteration of the outermost loop is running on an independent processor

```plaintext
for (i=1; i<=100; i++)
  for (j=1; j<=100; j++){
    X[i,j] = X[i,j] + Y[i-1, j]; /* S1 */
    Y[i,j] = Y[i,j] + X[i, j-1]; /* S2 */
  }

for (p=-100; p<=99; p++)
  for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
      if (p== i-j-1)
        X[i,j] = X[i,j] + Y[i-1, j]; /* S1 */
      if (p== i-j)
        Y[i,j] = Y[i,j] + X[i, j-1]; /* S2 */
    }
```
Affine Parallel Schedule

Affine Transformation for Parallelization
- Consider outermost loop as parallel schedule
- Each iteration of the outermost loop is running on an independent processor

```
for (i=1; i<=100; i++)
    for (j=1; j<=100; j++){
        X[i,j] = X[i,j] + Y[i-1, j]; /* S1 */
        Y[i,j] = Y[i,j] + X[i, j-1]; /* S2 */
    }
```
Loop Parallelization

• Where there is no dependence between loop iterations
  - Maximum amount of parallelism
  - How to check if there is dependence between loop iterations

• Where there is dependence between loop iterations
  - Divide loop iterations into multiple partitions
  - Synchronization Free
    • Each partition → one processor
    • Dependence only exists within one partition
  - Synchronize Necessary
    • Each partition → one time tick
    • Dependence only exists across partition
Review: Data Dependence

• Definition
  Given two memory references, there exists a dependence between them if the three following conditions hold:
  ✓ They reference the same array (cell)
  ✓ One of them is a write
  ✓ The two associated statements are executed

• Two memory accesses \(<F, f, B, b>\) and \(<F’, f’, B’, b’>\) are data dependent iff
  - At least one of them is a write reference and
  - There exist two vectors \(i\) and \(i'\) such that
    - \(B*i+b >= 0\)
    - \(B'i'+b'>=0\)
    - \(F*i + f = F'*i' + f'\).
  Note for instances of the same access, we need to add constraint \(i \neq i'\).
Dependence Polyhedron

• Two memory accesses \(<F_1, f_1, B_1, b_1, P_1, p_1>\) and \(<F_2, f_2, B_2, b_2, P_2, p_2>\) are data dependent at loop level \(l\) iff

  - At least one of them is a write reference and
  - There exist two vectors \(i_1\) and \(i_2\) such that
    • \(B_1 \cdot i_1 + b_1 \geq 0\)
    • \(B_2 \cdot i_2 + b_2 \geq 0\)
    • \(F_1 \cdot i_1 + f_1 = F_2 \cdot i_2 + f_2\)
    • \(P_1 \cdot i_1 + p_1 \geq P_2 \cdot i_2 + p_2\)

Access the same memory location
Within the loop bounds
Precedence condition

How to define precedence?
Precedence condition for a dependence at loop level \(l\), \(0 \leq l \leq d\) (\(d\) is nested-loop depth)

1. for \(k < l\), \(i_k = i'_k\)
2. for \(k = l\), \(i_k \leq i'_k\)
Consider dependence for reference \( n[j] \) for statement S2 at loop level \( i \)

\[
\begin{align*}
\text{for } (i = 1; i \leq 3; i++) \{ \\
\quad m[i] = i; \quad /* S1 */ \\
\quad \text{for } (j = 1; j \leq 3; j++) \\
\quad \quad n[j] = (n[j] + m[i])/2; \quad /*S2*/
\}
\end{align*}
\]

\((i_1, j_1), (i_2, j_2)\)

Two loop iterations

Access the same memory location

\[
\begin{bmatrix}
0 & 1 & 0 & -1 & 0 \\
1 & 0 & 0 & 0 & -1 \\
-1 & 0 & 0 & 0 & 3 \\
0 & 1 & 0 & 0 & -1 \\
0 & -1 & 0 & 0 & 3 \\
0 & 0 & 1 & 0 & -1 \\
0 & 0 & -1 & 0 & 3 \\
0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & -1 & 3 \\
-1 & 0 & 1 & 0 & -1
\end{bmatrix}
\]

Loop Bounds

Precedence condition

\[
\begin{bmatrix}
i_1 \\
i_2 \\
j_1 \\
j_2 \\
j_1
\end{bmatrix} \begin{bmatrix}
0 \\
= \\
> \\
= \\
0
\end{bmatrix}
\]
Program Dependence Graph (PDG)

1. Initialize PDG with one node for each statement in the program.

2. For each pair of statements R and S,
   For each pair of references $f_R$, $f_S$ in R and S
   For each loop level $l$,
   - Check if dependence polyhedron $D_{R,S, f_R, f_S, l}$ is empty
     - If it is not empty, then add an edge between statement R and statement S, annotate the edge with dependence information: type, direction, ref, and et.c
Program Dependence Graph Example

```
for ( i = 1; i <= 3; i++ ) {
    m[i] = i;      /* S1 */
    for ( j = 1; j <= 3; j++ )
        n[j] = (n[j] + m[i])/2;   /*S2*/
}
```
Affine Schedule for Time

Definition: Precedence Condition

Given \((T_R, t_R)\) a schedule for the statement \(R\), \((T_S, t_S)\) a schedule for the statement \(S\). \((T_R, t_R)\) and \((T_S, t_S)\) are legal schedules if for any two iterations \((i_R, i_S) \in D_{R,S}\)

\[ T_R \cdot i_R + t_R < T_S \cdot i_S + t_S \]

Notation:
- \(R\) refers to resource, \(S\) refers to sink, in the dependence notation.
Legal Affine Time Schedule

• A Naive Approach

1. For each pair of \((i_R, i_S) \in D_{R,S}\)
   Replace the \((i_R, i_S)\) with their exact values in
   
   \[ T_R*i_R + t_R + 1 \leq T_S*i_S + t_S \]

2. Solve for the affine coefficients of \((T_R, t_R)\) and \((T_S, t_S)\)

3. Challenge:
   - Solving the ILP for \(D_{R,S}\) is not easy
   - Even that is possible, might result in many pairs of dependent iterations in \(D_{R,S}\)
Farkas Lemma (Affine Form)

Let $D$ be a non-empty polyhedron defined by $p$ affine inequalities:

$$a_k^*x + b_k \geq 0, \ k = 1, p$$

Then affine form $\phi$ is nonnegative everywhere in $D$ iff it is a positive affine combination

$$\phi(x) = \lambda_0 + \sum_k \lambda_k (a_k^*x + b_k), \ \lambda_k \geq 0$$

The set of $\lambda_k$ coefficients are called Farkas Multipliers.
Farkas Lemma

• Applying Farkas Lemma

1. Time stamp for each iteration \( i \) is non-negative
\[
\theta(i) = T^*i + t \geq 0
\]
And \( \theta \) is defined over the loop iteration space
\[
B^*i + b \geq 0
\]
Therefore \( \theta(i) = \lambda^T(B^*i+b) + \lambda_0 \) and \( \lambda_k \geq 0, 0 \leq k \leq d \) (d-dimensional loop)

2. Precedence condition: the iteration \( i_2 \) must happen after \( i_1 \) if \( i_2 \) depends on \( i_1 \)
\[
T_2^*i_2 + t_2 - (T_1^*i_1 + t_1) \geq 0
\]
And it is defined over the dependence polyhedron)

• \( B_1^*i_1 + b_1 \geq 0 \)
• \( B_2^*i_2 + b_2 \geq 0 \)
• \( F_1^*i_1 + f_1 = F_2^*i_2 + f_2. \)
• \( P_2^*i_2 + p_2 \geq P_1^*i_1 + p_1. \)
Applying Farkas Lemma

\[
\begin{align*}
\theta_1(i_1) &= \lambda_1^T(B_1^*i_1 + b_1) + \lambda_{1,0}; \\
\theta_2(i_2) &= \lambda_2^T(B_2^*i_2 + b_2) + \lambda_{2,0}; \\
\theta_1(i_1) - \theta_2(i_2) &= \mu_1^T(B_1^*i_1 + b_1) + \mu_2^T(B_2^*i_2 + b_2) \\
&\quad + \mu_3^T(F_1^*i_1 + f_1 - F_2^*i_2 - f_2) \\
&\quad + \mu_4^T(F_2^*i_2 + f_2 - F_1^*i_1 - f_1) \\
&\quad + \mu_5^T(P_2^*i_2 + p_2 - P_1^*i_1 - p_1)
\end{align*}
\]

- \(B_1^*i_1 + b_1 \geq 0\)
- \(B_2^*i_2 + b_2 \geq 0\)
- \(F_1^*i_1 + f_1 = F_2^*i_2 + f_2\)
- \(P_2^*i_2 + p_2 \geq P_1^*i_1 + p_1\)
Eliminate loop indication variables

Eliminate the loop induction variables

\[
\lambda_1^T(B_1^*i_1 + b_1) + \lambda_{1,0} - \lambda_2^T(B_2^*i_2 + b_2) - \lambda_{2,0} = \\
\mu_1^T(B_1^*i_1 + b_1) + \mu_2^T(B_2^*i_2 + b_2) \\
+ \mu_3^T(F_1^*i_1 + f_1 - F_2^*i_2 - f_2) \\
+ \mu_4^T(F_2^*i_2 + f_2 - F_1^*i_1 - f_1) \\
+ \mu_5^T(P_2^*i_2 + p_2 - P_1^*i_1 - p_1)
\]

Step 1: Equate the coefficients of \( i \) variables and \( i \) variables are gone.

Step 2: Use Fourier-Motzkin elimination to eliminate as many Farkas multipliers as possible so that only affine schedule coefficients are left.

\begin{align*}
\text{• } & B_1^*i_1 + b_1 \geq 0 \\
\text{• } & B_2^*i_2 + b_2 \geq 0 \\
\text{• } & F_1^*i_1 + f_1 = F_2^*i_2 + f_2. \\
\text{• } & P_2^*i_2 + p_2 \geq P_1^*i_1 + p_1.
\end{align*}
for (i=0; i<=n; i++) {
    s(i) = 0;                                  /* S1 */
    for (j=0; j<=n; j++)
        s(i) = s(i) + a(i,j) * x(j);   /* S2 */
}

Schedules for S1 and S2:
\[ \theta(S1, i) = \mu_{1,0} + \mu_{1,1}i + \mu_{1,2}(n - i), \]
\[ \theta(S2, i, j) = \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n - i) + \mu_{2,3}j + \mu_{2,4}(n - j). \]

Two dependence edges in PDG:
• S1 depends on S2 when \( j = 0 \).
• S2(\( j' \)) depends on S2(\( j \)) when \( j' = j - 1. \) \( i' = i \)

For \((S1, S2)\) dependence edge in PDG (\( j = 0 \)):
\[ \theta(S2, i, j) - \theta(S1, i): \]
\[ \mu_{2,0} - \mu_{1,0} - 1 + (\mu_{2,1} - \mu_{2,2} - \mu_{1,1} + \mu_{1,2})i \]
\[ + (\mu_{2,3} - \mu_{2,4})j + (\mu_{2,2} + \mu_{2,4} - \mu_{1,2})n \]
\[ = \lambda_{1,0} + (\lambda_{1,1} - \lambda_{1,2})i + (\lambda_{1,3} - \lambda_{1,4} - \lambda_{1,5})j + (\lambda_{1,2} + \lambda_{1,4})n \]

\[ \mu_{2,0} - \mu_{1,0} - 1 = \lambda_{1,0} \]
\[ \mu_{2,1} - \mu_{2,2} - \mu_{1,1} + \mu_{1,2} = \lambda_{1,1} - \lambda_{1,2} \]
\[ \mu_{2,2} + \mu_{2,4} - \mu_{1,2} = \lambda_{1,2} + \lambda_{1,4} \]
\[ \mu_{2,3} = \lambda_{1,3} - \lambda_{1,4} - \lambda_{1,5} \]

For \((S2, S2)\) dependence edge in PDG
S2(\( i, j' \)) depends on S2(\( i, j \)) when \( j' = j - 1. \)
\[ \theta(S2, i, j) - \theta(S2, i, j-1) - 1 = \]
\[ \mu_{2,3} - \mu_{2,4} - 1 \geq 0 \]

Happens to contain only invariants, no variables.
No need for applying farkas lemma on dependence polyhedra.

Finally we have
\[ \mu_{2,0} \geq \mu_{1,0} + 1 \]
\[ \mu_{2,1} + \mu_{2,4} \geq \mu_{1,1} \]
\[ \mu_{2,2} + \mu_{2,4} \geq \mu_{1,2} \]
\[ \mu_{2,3} \geq \mu_{2,4} + 1 \]

All \( \lambda_{1,0} \) and variables are non-negative.
Use fourier-motzkin elimination.
Constraints on schedule coefficients

\[
\begin{align*}
\mu_{2,0} & \geq \mu_{1,0} + 1 \\
\mu_{2,1} + \mu_{2,4} & \geq \mu_{1,1} \\
\mu_{2,2} + \mu_{2,4} & > \mu_{1,2} \\
\mu_{2,3} & \geq \mu_{2,4} + 1
\end{align*}
\]

for (i=0; i<=n; i++) {
    s(i) = 0;                               /* S1 */
    for (j=0; j<=n; j++)
        s(i) = s(i) + a(i,j) * x(j);   /* S2 */
}

Affine schedules for S1 and S2:
\[
\begin{align*}
\theta(S1, i) &= \mu_{1,0} + \mu_{1,1}i + \mu_{1,2}(n - i), \\
\theta(S2, i, j) &= \mu_{2,0} + \mu_{2,1}i + \mu_{2,2}(n - i) + \mu_{2,3}j + \mu_{2,4}(n - j).
\end{align*}
\]
Code Generation

for (i=0; i<=n; i++) {
    s(i) = 0;    /* S1 */
    for (j=0; j<=n; j++)
        s(i) = s(i) + a(i,j) * x(j);   /* S2 */
}

θ(S1, i) = 0
θ(S2, i, j) = j +1

forall (i=0; i<=n; i++)
S(i) = 0;
for (j=0; j<=n; j++) {
forall (i=0; i<=n; i++) {
    s(i) = s(i) + a(i,j) * x(j);   /* S2 */
}
Code Generation

Legal schedule 2

\[ \theta(S1, i) = i \]
\[ \theta(S2, i, j) = i + j + 1 \]
for (i=0; i<=n; i++) {
    s(i) = 0;                               /* S1 */
    for (j=0; j<=n; j++)
        s(i) = s(i) + a(i,j) * x(j);   /* S2 */
}

s(0) = 0;                                                   /* S1 */
for (k = 1; k <= 2n+1; k++)
    forall (i=0; i<= k; i++) {
        if ( i == k && k <= n )
            s(k) = 0;                                          /* S1 */
        else if ( i < k)
            s(i) = s(i) + a(i, k-1-i) * x(k-1-i);      /* S2 */
    }

θ(S1, i) = i
θ(S2, i, j) = i + j +1

Legal schedule 2
The Optimal Schedule Problem

1. Introduce a new variable L such that
   \[ \theta(i) \leq L \]
2. By Farkas Lemma, we have
   \[ L - \theta(i) \geq 0 \]
   defined over the loop iteration polyhedron
   \[ B^*i + b \geq 0 \]
   Therefore \( L - \theta(i) \) can be expressed as a non-negative linear combination of
   \[ B^*i + b \geq 0 \]
3. We eliminate all the i variables, and obtain a polyhedron with respect to farkas multipliers (the schedule farkas multipliers and the L-related farkas multipliers). Then find minimum of L using ILP techniques.

Reading

- **Compilers — Principles, Techniques, Tools**
  - Chapters 11.1 – 11.4, 11.6-11.7

- **Papers**