Lecture 7: Parallelism and Dependence Analysis II
Review: Affine Loop Analysis

• A function $f$ of variables $x_1, x_2, ..., x_n$ is affine
  - If it is in such a form
    • $f = c_0 + c_1x_1 + c_2x_2 + ... + c_nx_n$, 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 + 2j - 1]

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

• Three 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: Loop Parallelization

• An Example

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

• Assumptions on loops
  - 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.

        for (i = 0; i <= 5; i++)
        for (j = i; j <= 7; j++)
            Z[ j, i] = 0;
Review: Iteration Spaces

• The conjunction (logical AND) of all linear equalities on loop bounds defines a **convex polyhedron**, which is the iteration space of the loop.
  - E.g.

    ```
    i>=0;  for (i=0; i<=5; i++)
    i<=5;  for (j=i; j<=7; j++)
    j>=i;  Z[j, i] = 0;
    j<=7;
    ```

• **A convex polyhedron**
  - If two points are in it, all points on the line between them are also in it.
Review: Lexicographic Order

- Order of sequential executions
  - Sweeping through the space in an ascending lexicographic order:
    \[ (a, b) \leq (a', b') \text{ iff } a < a' \text{ or } (a = a' \text{ & } b \leq b') \]

Lexicographic order for the vector \((i, j)\)
Review: Lexicographic Order

- Order of sequential executions
  - Sweeping through the space in an ascending lexicographic order:
    
    
    \[(a, b) \leq (a', b') \text{ iff } a < a' \text{ or } (a = a' \text{ and } b \leq b')\]

Lexicographic order for the vector \((j, i)\)
Review: Controlling Execution Order

• Question to answer
  - Given a re-ordering of the loop, how do we generate the loop bounds?

<table>
<thead>
<tr>
<th>for (i=0; i&lt;=5; i++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (j=i; j&lt;=7; j++)</td>
</tr>
<tr>
<td>Z[j, i] = 0;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>for (j=?; j&lt;=?; j++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=?; i&lt;=?; i++)</td>
</tr>
<tr>
<td>Z[j, i] = 0;</td>
</tr>
</tbody>
</table>

• Examples
  1. Suppose we want to vertically sweep through the space; the ordering is j, i.
     
     *What are the loop bounds?*

  2. Suppose we want to sweep through the space diagonally.
     
     *What are the loop bounds? Any other change to the loop induction variables?*
Review: Loop Interchange

- Project the polyhedron onto the outer dimension to determine the bounds of the outer loop.

```
for (i=0; i<=5; i++)
  for (j=i; j<=7; j++)
    Z[j, i] = 0;
```

for (j=?; j<=?; j++)
  for (i=?; i<=?; i++)
    Z[j, i] = 0;
Review: Projection

- Formal definition

The projection of an n-dimensional polyhedron S onto the first m of its dimensions is the set of points \((x_1, x_2, ..., x_m)\) such that for some \(x_{m+1}, x_{m+2}, ..., x_n\), vector \([x_1, x_2, ..., x_n]\) is in S.
Review: Compute Projection

- **Input:**
  A polyhedron $S$ defined by a set of linear constraints on $x_1, x_2, ..., x_n$. A given variable $x_m$ that is to be eliminated.

- **Output:**
  A polyhedron $S'$ defined by linear constraints on $x_1, x_2, ..., x_{m-1}, x_m + 1, ..., x_n$ that is a projection of $S$ onto dimensions other than the $m$-th.

- **Method:**
  *Fourier-Motzkin elimination.*
Review: Fourier-Motzkin Elimination

• To project to the j dimension.
  - Need to eliminate variable i.

\[ 0 \leq i, \& i \leq j \rightarrow 0 \leq j; \]
\[ j \leq 7. \]

```
for (i=0; i<=5; i++)
  for (j=i; j<=7; j++)
    Z[j, i] = 0;
```

\[ (i, j) \]
\[ i \geq 0; \]
\[ i \leq 5; \]
\[ j \geq i; \]
\[ j \leq 7; \]
Review: Fourier-Motzkin Elimination

• Algorithm:
  - For every pair of a lower bound and an upper bound on $x_m$, such as

    $$L \leq c_1 x_m \& c_2 x_m \leq U,$$

  create a new constraint

    $$c_2 L \leq c_1 U.$$

  - $S'$ is the set including all new constrains and those in $S$ that do not contain $x_m$.
  - It is possible that $S'$ is an empty space.
Review: Loop-Bounds Generation

- Compute the loop bounds from the innermost to the outer loops.

\[ S_n = S; \]
\[ \text{for } (i=n; i>=1; i--) \{
    L_{vi} = \text{all the lower bounds on } v_i \text{ in } S_i; \\
    U_{vi} = \text{all the upper bounds on } v_i \text{ in } S_i; \\
    S_{i-1} = \text{Constraints by eliminating } v_i \text{ from } S_i;
\}
\]

/* remove redundancies */
\[ S' = \Phi; \]
\[ \text{for } (i=1; i<=n; i++) \{
    \text{Remove any bounds in } L_{vi} \text{ and } U_{vi} \text{ implied by } S'; \\
    \text{Add the remaining constraints of } L_{vi} \text{ and } U_{vi} \text{ on } v_i \text{ to } S';
\}

\[
\begin{align*}
\text{for } (i=0; i<=5; i++) \\
& \text{for } (j=i; j<=7; j++) \\
& Z[j, i] = 0;
\end{align*}
\]

\[
\begin{align*}
\text{i} & \geq 0; \\
\text{i} & \leq 5; \\
\text{j} & \geq i; \\
\text{j} & \leq 7;
\end{align*}
\]

Target order: \((j,i)\)

\[
\begin{align*}
\text{L}_i & : 0 \\
\text{U}_i & : 5, j \\
\text{L}_j & : 0 \\
\text{U}_j & : 7
\end{align*}
\]

bounds on \(i\) is \((0, \min(5,j))\)

bounds on \(j\) is \((0, 7)\)
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;

Target: sweep through diagonally.

[0,0], [1,1], [2,2], [3,3], [4,4], [5,5]
[0,1], [1,2], [2,3], [3,4], [4,5]
[0,2], [1,3], [2,4], [3,5]
...
[0,6], [1,7]
[0,7]

k=j-i, order: k, j.

i = j-k>=0;
i = j-k<=5;
j >= j-k;
j <= 7.

L_j: k
U_j: 5+k, 7
L_k: 0
U_k: 7

for (k=0; k<=7; k++)
    for (j=k; j<=min(5+k,7); j++)
        Z[j, j-k] =0;

for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;

j>=i;
j<=7;
i>=0;
i<=5;
j>);
Data Space

• An Example

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

• An Example

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

f_W(i) = i+10    f_R(i) = i
```

![Diagram](image)
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
• \(<F, f, B, b>\) 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 vectors. (\(d\) for the # of loop levels)

\[
\begin{align*}
\text{Bi+b>0:} & & \text{Fi+f:} \\
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 0 \\
-1 & 1 & 0 \\
0 & -1 & 0 \\
0 & 0 & -1 \\
0 & 0 & -1 \\
\end{bmatrix} & & \begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix} \\
\text{for (i=0; i<=5; i++)} & & \text{for (j=i; j<=7; j++)} \\
& & \text{for (k=j; k<=9; k++)} \\
& & Z[j, i+1, 4] = 0;
\end{align*}
\]
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 if
  
  - At least one of them is a write reference and
  - There exist two vectors i and i’ such that
    
    \[ B*i+b \geq 0 \]
    \[ B'i'+b'>0 \]
    \[ F*i + f = F'*i' + f' \]
    
    Note for instances of the same access, we need to add constraint \( i \neq i' \).
Integer Linear Programming (ILP)

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

• **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
     - If failed, no data dependences, otherwise, continue.
  2. Use a set of heuristics to examine the inequalities.
     - If still not sure, continue.
  3. Branch-and-bound approach (a general ILP approach)

\[
\begin{align*}
1. & \quad B^*i + b \geq 0 \\
2. & \quad B'^*i' + b' \geq 0 \\
3. & \quad F^*i + f = F'^*i' + f' \\
    & \quad F^*i + f = F'^*i' + f' \\
    & \quad F^*i + f \leq F'^*i' + f' \\
    & \quad F^*i + f \geq F'^*i' + f'
\end{align*}
\]
Step 1: GCD Test

• Goal: check if there exists integer solutions to the equalities

\[ F^*i + f = F'^*i' + f' \]

**Diophantine** equation

• An equation with the condition that solutions must be integer.
• All equations in \( F^*i + f = F'^*i' + f' \) are linear Diophantine equations for dependence analysis.
Theorem

The linear Diophantine equation

\[ a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = c \]

has an integer solution \textbf{iff} \( \text{gcd}(a_1, a_2, \ldots, a_n) \) divides \( c \).

Example:

```plaintext
for (i=1; i<10; i++){
}
```

\( 2i = 2i' + 1 \)

\( \text{i.e.} \)

\( 2i - 2i' = 1. \)

\( \text{gcd}(2, -2) = 2. \) Not a divisor of 1.

No dependences.

GCD (Greatest Common Divisor)

- \( \text{gcd}(24, 36, 54) = 6. \)
More Examples

- $24x + 36y + 54z = 30$ Has integer solutions?
- $x - 2y + z = 0$ Has no integer solutions?
- $3x + 2y + z = 5$
Step 2

• **Approach 1: independent-variables test**
  - Applicable cases
    • The inequalities involving only one unknown
  - Just check whether integers exist between the upper and lower bounds

Example:

```c
for (i=0; i<= 10; i++)
  for (j=0; j<= 10; j++)
    Z[i,j] = Z[i+10, j+11];
```

```
0<= i, j, i’, j’ <=10
i = i’+10
j = j’ + 11
thus
0<= i <=10
0<= j <=10
0<= i’ = i-10 <=10
0<= j’ = j-11 <=10
```
Step 2

• **Approach 2: loop-residue test**
  - Solution
    • Build a directed graph
      • Nodes are variables
      • Edges from $v_i$ to $v_j$
      • Edge weight: $c$
  - If there is a cycle in the graph with negative total weight, no solutions to the inequalities.
    - special case
      • $v \leq c \quad \Rightarrow \quad v \leq v_0 + c$
      • $c \leq v \quad \Rightarrow \quad v_0 \leq v - c$
Step 2

• **Approach 2: loop-residue test**
  - Solution
    • Build a directed graph
    - Nodes are variables
    - Edges from $v_i$ to $v_j$
    - Edge weight: $c$

![](image.png)

```
1 <= v_1, v_2 <= 10
0 <= v_3 <= 4
v_2 <= v_1
2v_1 <= 2v_3 - 7
```
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:**

\[ S_n = S; \]

\[ \text{for } (i=n; i>=1; i--) \{ \]

\[ L_{v_i} = \text{all the lower bounds on } v_i \text{ in } S_i; \]

\[ U_{v_i} = \text{all the upper bounds on } v_i \text{ in } S_i; \]

\[ S_{i-1} = \text{Constraints by eliminating } v_i \text{ from } S_i; \]

\[ \} \]

/* remove redundancies */

\[ S' = \Phi; \]

\[ \text{for } (i=1; i<=n; i++) \{ \]

\[ \text{Remove any bounds in } L_{v_i} \text{ and } U_{v_i} \text{ implied by } S'; \]

\[ \text{Add the remaining constraints of } L_{v_i} \text{ and } U_{v_i} \text{ on } v_i \text{ to } S'; \]

\[ \} \]

---

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

\[ \{ \]

\[ \text{if (} S_i \text{ does not include an integer value) break; } \]

\[ \text{pick } c_i, \text{ an integer in the middle of the range for } v_i \text{ in } S_i; \]

\[ \text{modify } S_i \text{ by replacing } v_i \text{ by } c_i; \]

\[ \} \]

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 \( S_n \cup \{ v_i <= \text{floor}(l_i) \} \) and \( S_n \cup \{ v_i >= \text{ceiling}(u_i) \} \);

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

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

- Affine Transformation for Parallelization
  - Synchronization free
  - Synchronization needed
Affine Parallel Schedule

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

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
  • Synchronization free
  • Synchronization needed
Affine Parallel Schedule

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

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
  • Synchronization free
    • Synchronization needed
Parallelization & Space Partition

- To map each dynamic instance of a statement to a proc ID.
  - Constraints for synchronization free
    - Each pair of operations with dependences must be mapped to the same processor
  - Constraints for efficiency
    - Operations are not placed on the same proc unless necessary.

```c
for (i=1; i<=N; i++) {
    Y[i] = Z[i];
    X[i] = Y[i];
}
```

Assuming infinite processors; later can be mapped to physical proc.
Parallelization & Space Partition

To map each dynamic instance of a statement to a proc ID.

- Constraints for synchronization free
  - Each pair of operations with dependences must be mapped to the same processor
- Constraints for efficiency
  - Operations are not placed on the same proc unless necessary.

Assuming infinite processors; later can be mapped to physical proc.

```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 */
    }
```
Partition Schedule (at Space)

• \(<C, c>\) 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. \(p = C*i + c\)

• Examples

1-d processor grid

\[
\text{for } (i=1; i<=N; i++) \\
\ Y[i] = Z[i];
\]

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

2-d processor grid

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

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

Notation:
\textbf{bold fonts} for container variables; \textit{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. \(B_1*i_1 + b_1 \geq 0\), and
    2. \(B_2*i_2 + b_2 \geq 0\), and
    3. \(F_1*i_1 + f_1 = F_2*i_2 + f_2\),
   it must be the case that \(C_1*i_1 + c_1 = C_2*i_2 + c_2\).
• Goal: to find the partitions 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 $X[i,j]$ & $X[i,j-1]$:

(i$_1$, j$_1$): 1<=$i_1$<=$100$, 1<=$j_1$<=$100$,

(i$_2$, j$_2$): 1<=$i_2$<=$100$, 1<=$j_2$<=$100$,

$i_1 = i_2$, $j_1 = j_2 - 1$,

$$[C_{11} \quad C_{12}]^i_j + [c_1] = [C_{21} \quad C_{22}]^i_j + [c_2]$$

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

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

\[(i_3, j_3): 1 \leq i_3 \leq 100, \quad 1 \leq j_3 \leq 100,\]
\[(i_4, j_4): 1 \leq i_4 \leq 100, \quad 1 \leq j_4 \leq 100,\]
\[i_3 = i_4 - 1, \quad j_3 = j_4,\]

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{13} & C_{14}
\end{bmatrix}
\begin{bmatrix}
i_3 \\
j_3
\end{bmatrix}
+
\begin{bmatrix}
c_1 \\
c_2
\end{bmatrix}
=
\begin{bmatrix}
C_{21} & C_{22} \\
C_{23} & C_{24}
\end{bmatrix}
\begin{bmatrix}
i_4 \\
j_4
\end{bmatrix}
+
\begin{bmatrix}
c_1 \\
c_2
\end{bmatrix}
\]

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 */
}

\[Y[i,j] = Y[i, j] + X[i, j-1]; \quad /* S2 */\]
Step 2 Reduce Unknowns

Apply Gaussian Elimination to

\[
\mathbf{F}_1 \cdot i_1 + f_1 = \mathbf{F}_2 \cdot i_2 + f_2
\]

\[
\begin{align*}
1 \leq i_1 & \leq 100, \quad 1 \leq j_1 \leq 100, \\
1 \leq i_2 & \leq 100, \quad 1 \leq j_2 \leq 100, \\
i_1 = i_2, & \quad j_1 = j_2 - 1,
\end{align*}
\]

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

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

\[
\begin{bmatrix}
C_{11} - C_{21} & C_{12} - C_{22}
\end{bmatrix}
\begin{bmatrix}
i_1 \\
j_1
\end{bmatrix} + [c_1 - c_2 - C_{22}] = 0
\rightarrow
C_{11} - C_{21} = 0, \quad C_{12} - C_{22} = 0, \quad & c_1 - c_2 - C_{22} = 0.
\]

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

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

\[
C = \begin{bmatrix}
  C_{11} & C_{12} \\
  C_{21} & C_{22}
\end{bmatrix}
\]
\[
c = \begin{bmatrix}
  c_1 \\
  c_2
\end{bmatrix}
\]

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

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

Affine schedule for S2: \( C = [C_{21} \ C_{22}] = [1 \ -1], \ c = c_2 = 0 \)
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 */
  }

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

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

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

- Step 1: find useful processor ID ranges
  - S1: $-100 \leq p \leq 98$
  - S2: $-99 \leq p \leq 99$
  - Union: $-100 \leq p \leq 99$

- Step 2: generate code

```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 */

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 */
```

S1(i, j): assigned to processor p = i-j-1;
S2(i, j): assigned to processor p = i-j.
Code Optimization (will be discussed at next class)

```c
/*space 1*/
X[1,100]=X[1,100]+Y[0,100]; /*S1*/
/*space 2*/
for (p=-99; p<=98; p++){
  /*space 2a*/
  if (p>=0)
    Y[p+1,1] = Y[p+1,1] + X[p+1, 0]; /* S2 */
  /*space 2b*/
  for (i=max(1,p+2); i<=min(100,100+p); i++)
    X[i,i-p-1] = X[i,i-p-1] + Y[i-1, i-p-1]; /* S1 */
    Y[i,i-p] = Y[i,i-p] + X[i, i-p-1]; /* S2 */
  /*space 2c*/
  if (p<=-1)
    X[100+p,100] = X[101+p,100] + Y[101+p-1, 100]; /* S1 */
}
/*space 3*/
Y[100,1] = X[100,0] + Y[100,1]; /*S2*/
```

Affine Schedule for S1:  \[ C = [1 -1], \ c = -1 \]
i.e.  \( (i,j) \) of S1 to processor \( p = i-j-1 \);

Affine Schedule for S2:  \[ C = [1 -1], \ c = 0 \]
i.e.  \( (i,j) \) 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 */
  }

1. Partition the nodes (loop iterations) to assign them to each processor.
   Affine Space Partition Algorithm to compute Cs & cs.

2. Basic code generation.
   Add processor loops & predicate statements.

3. Code optimizations (next class)
   Update loop bounds through Fourier-Motzkin Elimination.
   Remove tests through space splitting.
Task Level Parallelism
Task Level Parallelism

Dependence/precedence graph
  ▶ Directed acyclic graphs (DAG)
  ▶ A node represents a task
  ▶ A directed edge represents precedence constraint
  ▶ Example: reduction operation

\[ S = \text{reduce}(A[1], A[2], \ldots, A[N]) \]
Task Level Parallelism

Dependence/precedence graph
- Directed acyclic graphs (DAG)
- A node represents a task
- A directed edge represents precedence constraint
- Example: loop with dependent iterations

![Dependence precedence graph illustration](image-url)
Directed Acyclic Graph

Nodes are computations
- May have multiple start nodes and end nodes
- A node may carry a weight

Practice: Is this a DAG?

(a)

(b)

(c)
Directed Acyclic Graph

Nodes are computations
▶ May have multiple start nodes and end nodes
▶ A node may carry a weight

Practice: Is this a DAG?
Schedule a DAG

$T_p$: time to perform computation with $p$ processors

- $T_1$: work (total # operations)
  - Time when the computation sequentially
- $T_\infty$: critical path / span
  - Time when parallelizing as much as possible

Lower bounds:

$$T_p \geq \frac{T_1}{p}, \quad T_p \geq T_\infty$$

Maximum parallelism:

- $T_1 / T_\infty$
- Linear speedup

$$\frac{T_p}{T_1} = \Theta(p)$$

$T_1 = 11$

$T_\infty = 5$
Computing Critical Path

Compute earliest start times of nodes

- Keep a value called start-time $S$ with each node
- Do a topological sort of the DAG
- For each node $n$ in topological order and for each predecessor $p$ of $n$

$$S_n = \max(S_n, S_p + w_p)$$

Complexity:
- $O(|V| + |E|)$
An Example of DAG Schedule

Space and time mapping (processor, start time)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>start</td>
<td>a</td>
<td>b</td>
<td>f</td>
<td>end</td>
</tr>
<tr>
<td>P1</td>
<td>c</td>
<td>e</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>d</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization is Necessary

Synchronization Assumptions

- How a processor know when it is safe to execute an operation?
  - Use barrier to ensure all processors run in lock-step in every time tick.
  - Data is visible in shared memory M at the next time tick (step).
Optimal schedule

Problem definition

- Given a DAG, $p$ processors, find the shortest possible schedule
- A well studied problem in computer science

Three restricted cases where the optimal solution can be found

- When the graph is a rooted directed tree, unit tasks execution time, for an arbitrary number of processors [Hu 1961]
- When the graph is a general DAG, unit task execution time, and the number of processors is 2. [Coffman and Graham 1971]
- When the graph is “interval ordered”, and all tasks have equal execution time [Papadimitriou and Yannakakis 1979]

Many heuristics available in literature:
Heuristic: list scheduling

Maintain a ready list $L_{\text{ready}}$

- At every time tick, choose $p$ ready operations (use a priority functions)
- Assign them to available processors
- Update ready list $L_{\text{ready}}$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>start</td>
<td>a</td>
<td>b</td>
<td>f</td>
<td>end</td>
</tr>
<tr>
<td>P1</td>
<td>c</td>
<td>e</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>d</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Precedence/Dependence Graphs

Programmers specify DAG
Example: series-parallel, use the fork-join constructs, which by default are acyclic.

```
QuickSort QS
Data : ( A , n ): A list A of length n
Result : A in sorted order
p ← pivot [ A ]
  low ( A ) ← \{ x | x ∈ A , x ≤ p \}
  high ( A ) ← \{ x | x ∈ A , x > p \}

fork
S1 ← QS ( low ( A ) , length ( low ( A )))
S2 ← QS ( high ( A ) , length ( high ( A )))
join
return S1 + S2
```
Precedence/Dependence Graphs

- **Programmers specify DAG**
  Example: series-parallel, use the fork-join constructs, which by default are acyclic.

- **Automatic compile-time dependence analysis**
  Analyze the read and writes to the same memory location (in our affine transformation example)

  However, we cannot always determine precisely whether a dependence exist.
  (a) aliasing — two names that refer to the same storage, pointers and references
  (b) control flow — dependence may exist for some invocations and not for others

Potential solutions:
- **Conservative approximation**, assume dependence exist when in doubt, may loose some parallelization opportunity.

  OR we can **speculatively parallelize** by using extra cores in the multi-core/many-core processor.
Project 2 — Single Source Shortest Path (SSSP)

Finding shortest path from one node to another

- Dijkstra (1959)
- Bellman-Ford (1957)
- Delta-stepping (1998)
Dijkstra’s Algorithm (Sequential)

Prefer nodes that have smaller distance label
  - One node at most visited once (but might be updated multiple times)

while $Q$ is not empty
  $u \leftarrow$ vertex in $Q$ with min $\text{dist}[u]$
  remove $u$ from $Q$
  for each neighbor $v$ of $u$:  //where $v$ is still in Q.
    alt $\leftarrow$ $\text{dist}[u] + \text{length}(u, v)$;
    if alt $<$ $\text{dist}[v]$;
      //A shorter path to $v$ has been found
      $\text{dist}[v] \leftarrow$ alt;
      $\text{prev}[v] \leftarrow u$;
  return $\text{dist}[\ ]$, $\text{prev}[\ ]$;
Dijkstra’s Algorithm (Sequential)

- Example

```
Step 0: Initialization.

<table>
<thead>
<tr>
<th>v</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[v]</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
```
Dijkstra’s Algorithm (Sequential)

• Example

Step 1: As $Adj[s] = \{a, b\}$, work on $a$ and $b$ and update information.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$s$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d[v]$</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm (Sequential)

• Example

Step 2: After Step 1, \( a \) has the minimum key in the priority queue. As \( Adj[a] = \{b, c, d\} \), work on \( b, c, d \) and update information.

<table>
<thead>
<tr>
<th>( v )</th>
<th>( s )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d[v] )</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm (Sequential)

• Example

Step 3: After Step 2, $b$ has the minimum key in the priority queue. As $\text{Adj}[b] = \{a, c\}$, work on $a$, $c$ and update information.

<table>
<thead>
<tr>
<th>$v$</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d[v]$</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm (Sequential)

• Example

Step 4: After Step 3, c has the minimum key in the priority queue. As $\text{Adj}[c] = \{d\}$, work on d and update information.

<table>
<thead>
<tr>
<th>$v$</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d[v]$</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm (Sequential)

- Example

Step 5: After Step 4, \( d \) has the minimum key in the priority queue. As \( Adj[d] = \{c\} \), work on \( c \) and update information.

<table>
<thead>
<tr>
<th>( v )</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d[v] )</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Bellman-ford Algorithm (Parallel Version)

- **Example**

![Diagram of a graph with nodes V0, V1, V2, V3, V4, and V5.]

**Implementation 1**

<table>
<thead>
<tr>
<th>Node values</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>0</td>
</tr>
<tr>
<td>V1</td>
<td>4</td>
</tr>
<tr>
<td>V2</td>
<td>2</td>
</tr>
<tr>
<td>V3</td>
<td>10</td>
</tr>
<tr>
<td>V4</td>
<td>3</td>
</tr>
<tr>
<td>V5</td>
<td>11</td>
</tr>
</tbody>
</table>

**Edge list E:**

- 0 1 4
- 0 2 2
- 1 2 5
- 1 3 10
- 2 4 3
- 3 5 11
- 4 3 4

**Vertex values:**

- V[0]: 0
- V[1]: ∞
- V[2]: ∞
- V[3]: ∞
- V[4]: ∞
- V[5]: ∞

**Listing 1: Sequential Bellman-ford**

```java
for each vertex v in vertices:
    distance[v] := inf;
for i from 1 to size(vertices) - 1 {
    for each edge (u, v) with weight w in edges {
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w;
    }
    if (no node's distance changed) break;
}
```
Bellman-ford Algorithm (Parallel Version)

• Example

**Initialization**

| Node values | 0  | ∞  | ∞  | ∞  | ∞  | ∞  |

**Edge list E:**

| 0 1 4 |
| 0 2 2 |
| 1 2 5 |
| 1 3 10 |
| 2 4 3 |
| 3 5 11 |
| 4 3 4 |

<table>
<thead>
<tr>
<th>Vertex value V (initially):</th>
</tr>
</thead>
</table>

| V[0]: 0 |
| V[1]: ∞ |
| V[2]: ∞ |
| V[3]: ∞ |
| V[4]: ∞ |
| V[5]: ∞ |

**Listing 1: Sequential Bellman-ford**

```plaintext
for each vertex v in vertices:
    distance[v] := inf;
3         distance[source] = 0;
4   for i from 1 to size(vertices) - 1 {
5         for each edge (u, v) with weight w in edges {
6             if distance[u] + w < distance[v]:
7                 distance[v] := distance[u] + w;
8             }
9         if (no node’s distance changed) break;
10    }
```
Bellman-ford Algorithm (Parallel Version)

• Example

![Graph Diagram]

**Implementation 1**

<table>
<thead>
<tr>
<th>Node values</th>
<th>Initialization</th>
<th>Iteration 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V1</td>
<td>∞</td>
<td>4</td>
</tr>
<tr>
<td>V2</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>V3</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>V4</td>
<td>∞</td>
<td>5</td>
</tr>
<tr>
<td>V5</td>
<td>∞</td>
<td>20</td>
</tr>
</tbody>
</table>

**Edge list E:**

- 0 1 4
- 0 2 2
- 1 2 5
- 1 3 10
- 2 4 3
- 3 5 11
- 4 3 4

**Vertex value**

- V[0]: 0
- V[1]: ∞
- V[2]: ∞
- V[3]: ∞
- V[4]: ∞
- V[5]: ∞

**Listing 1: Sequential Bellman-ford**

```python
for each vertex v in vertices:
    distance[v] := inf;
    distance[source] = 0;
for i from 1 to size(verticies)-1 {
    for each edge (u, v) with weight w in edges {
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w;
    }
    if (no node's distance changed) break;
}
```
Bellman-ford Algorithm (Parallel Version)

• Example

**Implementation 1**

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Node values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Node values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 2</th>
<th>Node values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Edge list E:**
- 0 1 4
- 0 2 2
- 1 2 5
- 1 3 10
- 2 4 3
- 3 5 11
- 4 3 4

**Vertex value**
- \( V(0) \): 0
- \( V(1) \): ∞
- \( V(2) \): ∞
- \( V(3) \): ∞
- \( V(4) \): ∞
- \( V(5) \): ∞

**Listing 1: Sequential Bellman-ford**

```python
for each vertex \( v \) in vertices:
    distance[v] := inf;
    distance[source] = 0;
for i from 1 to size(vertices) - 1 {
    for each edge \( (u, v) \) with weight \( w \) in edges {
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w;
    }
    if (no node's distance changed) break;
}
```
Bellman-ford Algorithm (Parallel Version)

- Example

**Implementation 1**

**Node values**

<table>
<thead>
<tr>
<th>Initialization</th>
<th>0</th>
<th>∞</th>
<th>∞</th>
<th>∞</th>
<th>∞</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>5</td>
<td>25</td>
</tr>
</tbody>
</table>

**Edge list E:**

- 0 1 4
- 0 2 2
- 1 2 5
- 1 3 10
- 2 4 3
- 3 5 11
- 4 3 4

**Vertex value V (initially):**

- V[0]: 0
- V[1]: ∞
- V[2]: ∞
- V[3]: ∞
- V[4]: ∞
- V[5]: ∞

**Listing 1: Sequential Bellman-ford**

```c
1 for each vertex v in vertices:
2    distance[v] := inf;
3    distance[source] = 0;
4 for i from 1 to size(verticies)-1 {
5    for each edge (u, v) with weight w in edges {
6        if distance[u] + w < distance[v]:
7            distance[v] := distance[u] + w;
8        }
9    if (no node's distance changed) break;
10 }
```
Bellman-ford Algorithm (Parallel Version)

• Example

![Diagram of a graph with nodes V0 to V5 and edges labeled with weights.]

**Implementation 1**

**Node values**

<table>
<thead>
<tr>
<th>Initialization</th>
<th>0</th>
<th>∞</th>
<th>∞</th>
<th>∞</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Iteration 4</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

**Edge list E:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Vertex value V (initially):**

| V[0]: 0 |
| V[1]: ∞ |
| V[2]: ∞ |
| V[3]: ∞ |
| V[4]: ∞ |
| V[5]: ∞ |

**Listing 1: Sequential Bellman-ford**

```
1   for each vertex v in vertices:
2       distance[v] := inf;
3   distance[source] = 0;
4   for i from 1 to size(vertices) – 1 {
5       for each edge (u, v) with weight w in edges {
6           if distance[u] + w < distance[v]:
7               distance[v] := distance[u] + w;
8           
9       }
10   if (no node’s distance changed) break;
```
Reading

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

• **Papers**