Class Information

• Homework 7 is released.
• Homework 8 (our last homework) will be released this Wednesday.
• Project 2 is due this Thursday.
• Please pick up your midterm exam if you haven’t.
  You can pick them up at my office hour or any TA’s office hour.
A PROCESS or THREAD is a potentially-active execution context. Classic von Neumann model of computing has single thread of control, however parallel programs have more than one. A process or thread can be thought of as an abstraction of a physical PROCESSOR. Processes/Threads can come from multiple CPUs, kernel-level multiplexing of single physical machine, and language or library level multiplexing of kernel-level abstraction. They can run in true parallel, unpredictably interleaved, and run-until-block.
The dependence relation can be modeled as a directed graph such that if $A \rightarrow B$, the result of task $A$ is required for the processing of task $B$.

Dependence relation: all task-to-task execution orderings that must be preserved if the meaning of the program is to remain the same.

The dependence relation can be modeled as a directed graph such that if $A \rightarrow B$, the result of task $A$ is required for the processing of task $B$.

Example:

$S_1$: $\pi = 3.14$

$S_2$: $R = 5$

$S_3$: $\text{Area} = \pi \times R^2$

Statement-level dependence graph
Dependence Graph

• Directed acyclic graph (DAG)
• A node represents a task
• A directed edge represents precedence constraint

DAG example 1:
Dependence Graph

- Directed acyclic graph (DAG)
- A node represents a task
- A directed edge represents precedence constraint

DAG example 2:

\[ S = \text{sum}(A[1], A[2], ..., A[N]) \]

\[ \begin{align*}
+ & \rightarrow + \rightarrow + \rightarrow \cdots \rightarrow + \\
S &
\end{align*} \]
Scheduling a DAG

$T_p$: time to perform computation with $p$ processors
- $T_1$: work (total # operations)
- $T_\infty$: critical path or span

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

Maximum parallelism: $T_1 / T_\infty$

Linear speedup: \[ \frac{T_p}{T_1} = \Theta(p) \]
Scheduling a DAG

\( T_p \): time to perform computation with \( p \) processors

- \( T_1 \): work (total # operations)
- \( T_\infty \): critical path or span

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

Maximum parallelism: \( T_1 / T_\infty \)

Linear speedup: \( \frac{T_p}{T_1} = \Theta(p) \)

\( T_1 = ? \)
Scheduling a DAG

$T_p$: time to perform computation with $p$ processors
  - $T_1$: work (total # operations)
  - $T_\infty$: critical path or span

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

Maximum parallelism: $T_1 / T_\infty$

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

$T_\infty = ?$
Scheduling a DAG

$T_p$: time to perform computation with $p$ processors
  - $T_1$: work (total # operations)
  - $T_\infty$: critical path or span

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

Maximum parallelism: $T_1 / T_\infty$

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

$T_\infty = ?$
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called $S(n)$ associated with each node $n$
- For each node $n$
  
  $S(n)$ is the maximum of $\{ S(p) + 1 \}$, for all $p \in \text{pred}(n)$

Assuming a task takes 1 unit time
Computing Critical Path

Compute the earliest start time of each node

• Keep a value called $S(n)$ associated with each node $n$
• For each node $n$
  
  $S(n)$ is the maximum of \{ $S(p) + 1$ \}, for all $p \in \text{pred}(n)$

Assuming a task takes 1 unit time

![Diagram of a project network with nodes a, b, c, d, e, f, g, h, i, start, and end, showing dependencies and time units.]
Computing Critical Path

Compute the earliest start time of each node
• Keep a value called $S(n)$ associated with each node $n$
• For each node $n$
  $S(n)$ is the maximum of $\{ S(p) + 1 \}$, for all $p \in \text{pred}(n)$

Assuming a task takes 1 unit time
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called $S(n)$ associated with each node $n$
- For each node $n$
  
  $S(n)$ is the maximum of $\{ S(p) + 1 \}$, for all $p \in \text{pred}(n)$

Assuming a task takes 1 unit time

![Graph showing start and end nodes with intermediate nodes and their durations](image-url)
Computing Critical Path

Compute the earliest start time of each node

• Keep a value called $S(n)$ associated with each node $n$

• For each node $n$

  $S(n)$ is the maximum of \{ $S(p) + 1$ \}, for all $p \in \text{pred}(n)$

Assuming a task takes 1 unit time
Computing Critical Path

Compute the earliest start time of each node

• Keep a value called $S(n)$ associated with each node $n$

• For each node $n$

  $S(n)$ is the maximum of \{ $S(p) + 1$ \}, for all $p \in \text{pred}(n)$

Assuming a task takes 1 unit time
Based on if the dependence constraints have been resolved

- Schedule the nodes that are ready at every time tick
- A completed operation at the end of one time step can lead to more ready operations at next time tick

**Four threads T1, T2, T3, T4**

<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>T1 start</td>
<td>a</td>
<td>b</td>
<td>f</td>
<td>end</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>c</td>
<td>e</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>d</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming a task takes 1 unit time
We will use **loop analysis** as an example to describe automatic dependence analysis and parallelization.

**Assumptions:**

1. We only have scalar and subscripted variables (no pointers and no control dependence) for loop dependence analysis.

2. We focus on **affine loops**: both loop bounds and memory references are affine functions of loop induction variables.

A function \( f(x_1, x_2, \ldots, x_n) \) is **affine** if it is in such a form:

\[
f = c_0 + c_1 x_1 + c_2 x_2 + \ldots + c_n x_n,
\]

where \( c_i \) are all constants.
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
```
float Z[100];
for (i=0; i<10; i++)
    Z[i+10] = Z[i];
```

**Three Spaces**

- **Iteration space, data space, and processor space**

**Data Space**

**Iteration space**

**Processor space**

**Array Z[]**

Assuming one task is one loop iteration, what is the maximum parallelism?

Maximum parallelism: $T_1 / T_\infty$
Dependence Definition

**Bernstein’s Condition:** — There is a data dependence from statement (instance) $S_1$ to statement $S_2$ (instance) if

- Both statements (instances) access the same memory locations
- One of them is a write
- There is a run-time execution path from $S_1$ to $S_2$

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

No dependence across any two loop iterations!
Data Dependence Classifications

“S₂ depends on S₁” — (S₁ δ S₂)

**True (flow) dependence**
occurs when S₁ writes a memory location that S₂ later reads (RAW).

**Anti dependence**
occurs when S₁ reads a memory location that S₂ later writes (WAR).

**Output dependence**
occurs when S₁ writes a memory location that S₂ later writes (WAW).

**Input dependence**
occurs when S₁ reads a memory location that S₂ later reads (RAR).
Simple Dependence Testing

• Examples:

```c
for (i = 1; i <= 100; i++) {
    S1: A[i] = ...
    S2: ...= A[i - 1]
}
```

```c
float Z[100];
for (i =0; i < 12; i++) {
    S: Z[ i+10 ] = Z[i];
}
```

1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?
Dependence Testing

Single Induction Variable (SIV) Test

• Single loop nest with constant lower (LB) and upper (UB) bound, and step 1.

```
for i = LB, UB, 1
    ...
endfor
```

• Two array references as affine function of loop induction variable

```
for i = LB, UB, 1
    R1: X(a*i + c1) = ...
    R2: ... = X(a*i + c2) ...
endfor
```

Question: Is there a true dependence between R1 and R2?
There is a dependence between R1 and R2 iff

\[ \exists i, i': \ LB \leq i \leq i' \leq UB \text{ and } (a*i+c_1) = (a*i'+c_2) \]

where \( i \) and \( i' \) represent two iterations in the iteration space. This means that in both iterations, the same element of array \( X \) is accessed.

So let’s just solve the equation:

\[ (a * i + c_1) = (a * i' + c_2) \quad \Rightarrow \quad (c_1 - c_2)/a = i' - i = \Delta d \]

There is a dependence iff

- \( \Delta d \) is an integer value
- \( UB - LB \geq \Delta d \geq 0 \)
Simple Dependence Testing

• Examples:

```c
for (i = 1; i <= 100; i++) {
    S1:  A[i] = ...
    S2:  ...= A[i - 1]
}
```

```c
float Z[100];
for (i =0; i < 12; i++) {
    S:  Z[i+10] = Z[i];
}
```

1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?
Lexicographical Order

- Order of sequential loop executions
- Sweeping through the space in an ascending lexicographic order: 
  \((i, j) \leq (i', j')\) iff one of the two conditions is satisfied
  1. \(i < i'\)
  2. \(i = i' \land j \leq j'\)

\[
\begin{align*}
&\text{for (i = 1; i \leq 5; i++)} \\
&\quad \text{for (j = 1; j \leq 6 - i; j++)} \\
&\quad Z[j, i] = 0;
\end{align*}
\]
Dependence Testing

• Example:

\[
\begin{align*}
d \text{do } & I = 1, 99 \\
& \text{do } J = 1, 100 \\
& \quad A(I,J) = A(I+1,J) + 1 \\
& \text{end do} \\
& \text{end do}
\end{align*}
\]

1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?
4. Which loop (i or j) can be parallelized?
Next Class

Reading:

• ALSU, Chapter 11.1 - 11.3
Given

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

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

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

Does \( \exists \alpha \leq \beta \) in the loop iteration space, s.t.

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