An Example (a)

FOR i := 2 TO 7 DO
a[i] := a[i] + c; b[i] := a[i-1] * b[i];
ENDFOR

<table>
<thead>
<tr>
<th>i</th>
<th>Time</th>
<th>Statement</th>
</tr>
</thead>
</table>

An Example (b)

Schedule the iterations of the following loop onto three CPUs ($P_1$, $P_2$, $P_3$) using cyclic scheduling.

FOR i := 2 TO 7 DO
  $S_1$: a[i] := a[i] + c;
  $S_2$: b[i] := a[i-1] * b[i];
ENDFOR

<table>
<thead>
<tr>
<th>CPU</th>
<th>i</th>
<th>$S_1$</th>
<th>$S_2$</th>
</tr>
</thead>
</table>

An Example (c)


Hence, $P_2$ will be using the old (wrong) value of $a[5]$. 

P1

\[
\begin{array}{l}
\end{array}
\]

P2

\[
\begin{array}{l}
\end{array}
\]

P3

\[
\begin{array}{l}
\end{array}
\]

Wrong value of $a[5]!$
**Parallelizing Options I**

- **Approaches to fixing the problem:**
  1. Give up, and run the loop serially on one CPU.
  2. Rewrite the loop to make it parallelizable.
  3. Insert synchronization primitives.

  **Give up**

  We should notify the programmer why the loop could not be parallelized, so maybe he/she can rewrite it him/herself.

  **Rewrite the loop**

  ```pascal
  FOR i := 2 TO 7 DO
    S1: a[i] := a[i] + c;
    S2: b[i] := a[i-1]*b[i];
  ENDFOR
  ```

**Parallelizing Options II**

- **Synchronize w/ Event Counters**

  ```pascal
  VAR ev : EventCounter;
  FOR i := 2 TO 7 DO
    S1: a[i] := a[i] + c;
        advance(ev); await(ev, i-1)
    S2: b[i] := a[i-1]*b[i];
  ENDFOR
  ```

  - `await/advance` implements an **ordered critical section**, a region of code that the Workers must enter in some particular order.
  - `await/advance` are implemented by means of an **event counter**, an integer protected by a lock.
  - `await(ev, i)` sleeps until the event counter reaches `i`.
  - `advance(ev)` increments the counter.

**Parallelizing Options III**

- **Synchronize w/ Vectors**

  ```pascal
  VAR ev : SynchronizationVector;
  FOR i := 2 TO 7 DO
    S1: a[i] := a[i] + c;
    ev[i] := 1;
    IF i > 2 THEN
      wait(ev[i-1])
    ENDIF;
    S2: b[i] := a[i-1]*b[i];
  ENDFOR
  ```

  - `ev` is a vector of bits, one per iteration. It is protected by a lock and initialized to all 0's.
  - `wait(ev[i])` will sleep the process until `ev[i]=1`.
  - Initialization of the vector can be expensive.
What does a real compiler do?

Let's see how \texttt{pca} treats this loop.
\texttt{pca -unroll=1 -cmp -lo=cklnps -list=l.l l.c}

\texttt{C Program in l.c}

\begin{verbatim}
int i,n; double a[10000], b[10000];
main () {
    for(i=2; i<=n; i++) {
        a[i] = a[i] + 100.0;
        b[i] = a[i-1]*b[i]; }
}\end{verbatim}

\texttt{Listing in l.l}

for i
Original loop split into sub-loops
1. Concurrent
2. Concurrent
1 loops concurrentized

\texttt{Parallelized program in l.m}

\begin{verbatim}
int main( ) {
    int K1, K3;
    K3 = ((n - 1)>(0) ? (n - 1) : (0));
    #pragma parallel if(n > 51) byvalue(n)
        shared(a, b) local(K1) {
            #pragma pfor iterate(K1=2;n-1;1)
                for ( K1 = 2; K1<=n; K1++ )
                    a[K1] = a[K1] + 100.e0;
            #pragma synchronize
            #pragma pfor iterate(K1=2;n-1;1)
            for ( K1 = 2; K1<=n; K1++ )
                b[K1] = a[K1-1] * b[K1];
    }
    i = K3 + 2;
}\end{verbatim}

\texttt{pca's Choices II (a)}

\begin{verbatim}
for(i=2; i<=n; i++) {
    a[i] = a[i+1] + 100.0;
    b[i] = a[i-1]*b[i]; }
}\end{verbatim}

\texttt{Listing in d.l}

for i
Original loop split into sub-loops
1. Scalar
   Data dependence involving this
   line due to variable "a"
2. Concurrent
   1 loops concurrentized
Parallelized program in d.m

```c
for ( K1 = 2; K1<=n; K1++ )
    a[K1] = a[K1+1] + 100.0;
#pragma parallel if(n > 102) byvalue(n)
    shared(a, b) local(K1)
{
    #pragma pfor iterate(K1=2;n-1;1)
        for ( K1 = 2; K1<=n; K1++ )
            b[K1] = a[K1-1] * b[K1];
}
```

- This time pca
  1. split the loop in two subloops (like before),
  2. parallelized the second subloop, and
  3. gave up on the first subloop, executing it serially.

Concurrentization

A loop can be concurrentized iff all its data dependence directions are =.
In other words, a loop can be concurrentized iff it has no loop carried data dependences.

The I-loop below cannot be directly concurrentized. The loop dependences are $S_1 \delta =, < S_1$, $S_1 \delta =, = S_2$, $S_2 \delta <, = S_3$. Hence, the I-loop's dependence directions are (=, =, <).

```
FOR I := 1 TO N DO
    FOR J := 2 TO N DO
        S2: C[I,J] := A[I,J] + D[I+1,J];
        S3: D[I,J] := 0.1;
    END;
END;
```

Exam I (415.730/96)

```
FOR i := 1 TO n DO
    FOR j := 1 TO n DO
    END;
END;
```

1. Which of the dependencies are loop-carried?
2. Which of the loops can be directly concurrentized (i.e., run in parallel without any loop transformations or extra synchronization)? Motivate your answer!
3. What is the difference between a pre-scheduled and a self-scheduled loop? Under what circumstances should we prefer one over the other?
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.

Summary II

- When faced with a loop
  
  \[
  \text{FOR } i := \text{From TO To DO} \\
  S_1: \quad A[f(i)] := \ldots \\
  S_2: \quad \ldots \quad := A[g(i)] \\
  \text{ENDFOR}
  \]

  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:

  - The compiler decides that \( f(i) \) and \( g(i) \) are too complicated to analyze. ⇒ Run the loop serially.
  - 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.