CSc 553 — Principles of Compilation

37 : Parallelizing Compilers II

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

April 20, 2011

1 An Example (a)

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

i	Time	Statement
2	①	a[2]:=a[2]+c
	②	b[2]:=a[1]*b[2]
3	③	a[3]:=a[3]+c
	④	b[3]:=a[2]*b[3]
4	⑤	a[4]:=a[4]+c
	⑥	b[4]:=a[3]*b[4]
5	⑦	a[5]:=a[5]+c
	⑧	b[5]:=a[4]*b[5]
6	⑨	a[6]:=a[6]+c
	Ⓐ	b[6]:=a[5]*b[6]
7	Ⓑ	a[7]:=a[7]+c
	Ⓒ	b[7]:=a[6]*b[7]

2 An Example (b)

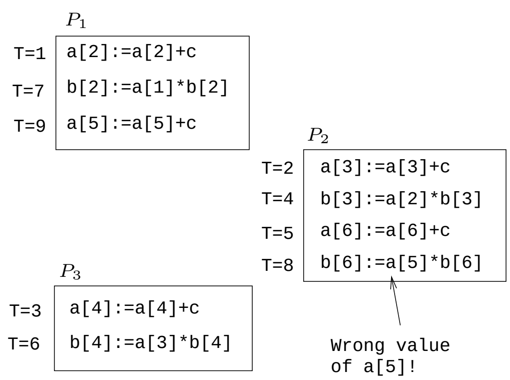
- Schedule the iterations of the following loop onto three CPUs (P_1, P_2, P_3) using cyclic scheduling.

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

CPU	i	S_1	S_2
P_1	2	a[2]:=a[2]+c	b[2]:=a[1]*b[2]
	5	a[5]:=a[5]+c	b[5]:=a[4]*b[5]
P_2	3	a[3]:=a[3]+c	b[3]:=a[2]*b[3]
	6	a[6]:=a[6]+c	b[6]:=a[5]*b[6]
P_3	4	a[4]:=a[4]+c	b[4]:=a[3]*b[4]
	7	a[7]:=a[7]+c	b[7]:=a[6]*b[7]

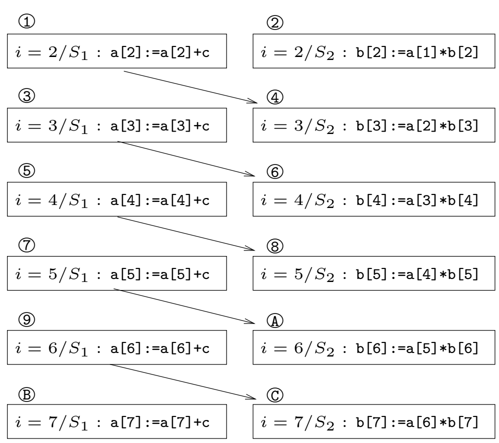
3 An Example (c)

- The three CPUs run asynchronously at different speeds. So, when P_2 is executing `b[6]:=a[5]*b[6]` at time `T=8`, P_1 has yet to execute `a[5]:=a[5]+c`.
- Hence, P_2 will be using the old (wrong) value of `a[5]`.



4 An Example (d)

- Statement $i/S_1:$ `a[i]:=a[i]+c` must run before statement $i+1/S_2:$ `b[i]:=a[i-1]*b[i]` in the next iteration.



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

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

6 Parallelizing Options II

Synchronize w/ Event Counters

```
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.

7 Parallelizing Options III

Synchronize w/ Vectors

```
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.

8

What does a real compiler do?

9 pca's Choices I (a)

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

C Program in `l.c`

```
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]; }}
```

Listing in `l.l`

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

10 pca's Choices I (b)

Parallelized program in `l.m`

```
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;
}
```

11 pca's Choices II (a)

- Let's try a slightly different loop....

C Program in d.c

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

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
```

12 pca's Choices II (b)

Parallelized program in d.m

```
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.

13

Concurrentization

14 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 $S_1\ \delta_{=,<}\ S_1$, $S_1\ \delta_{=,=}\ S_2$, $S_2\ \overline{\delta}_{<,=}\ S_3$. Hence, the I-loop's dependence directions are $(=,=,<)$.

```
FOR I := 1 TO N DO
   FOR J := 2 TO N DO
      S1:   A[I,J] := A[I,J-1] + B[I,J];
      S2:   C[I,J] := A[I,J] + D[I+1,J];
      S3:   D[I,J] := 0.1;
   ENDFOR
ENDFOR
```

15 Exam I (415.730/96)

```
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO
S1:         A[i,j] := A[i,j-1] + C;
         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?

16 Readings and References

- Padua & Wolfe, *Advanced Compiler Optimizations for Supercomputers*, CACM, Dec 1996, Vol 29, No 12, pp. 1184–1187.

17 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.

18 Summary II

- When faced with a loop

  ```
  FOR i := From TO To DO
     S1:   A[f(i)] := ···
     S2:   ··· := A[g(i)]
  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:

1. The compiler decides that $f(i)$ and $g(i)$ are too complicated to analyze. $\Rightarrow$ Run the loop serially.
2. 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. $\Rightarrow$ Run the loop in parallel or serially, depending on the outcome.