# CSE 4351/5351 Notes 8: Parallelizing Compilers, Sequent FORTRAN, and OpenMP

Elementary Data Dependence Concepts

```
  1: Entry
  1: integer a
  1: integer b
  1: integer c
  1: integer d
  1: integer e
  2: a = b+c
  3: d = a+2
  4: e = a+3
```

flow dependence 2: --> 3:
flow dependence 2: --> 4:

```
  1: Entry
  1: integer a
  1: integer b
  1: integer c
  1: integer d
  2: a = b+c
  3: b = d/2
```

anti dependence 2: --> 3:

```
  1: Entry
  1: integer a
  1: integer b
  1: integer c
  1: integer d
  1: integer e
  1: integer f
  2: a = b+c
  3: d = a+2
  4: a = e+f
```

flow dependence 2: --> 3:
output dependence 2: --> 4:
anti dependence 3: --> 4:

Loops and Data Dependence

```
  1: Entry
  1: integer a(1:3)
  1: integer b(1:3)
  1: integer c(1:3,1:2)
  2: for i = 1,3 do
  3:  a(i) = b(i)
  4:  for j = 1,2 do
  5:   c(i,j) = a(i)+b(j)
  4:  endfor
  2: endfor
```

May be "unrolled" as:

```
  1: Entry
  1: integer a(1:3)
  1: integer b(1:3)
  1: integer c(1:3,1:2)
  2: a(1) = b(1)
  3: c(1,1) = a(1)+b(1)
  4: c(1,2) = a(1)+b(2)
  5: a(2) = b(2)
  6: c(2,1) = a(2)+b(1)
  7: c(2,2) = a(2)+b(2)
  8: a(3) = b(3)
  9: c(3,1) = a(3)+b(1)
 10: c(3,2) = a(3)+b(2)
```

To yield the detailed dependence list:

flow dependence 2: --> 3:
flow dependence 2: --> 4:
flow dependence 5: --> 6:
flow dependence 5: --> 7:
flow dependence 8: --> 9:
flow dependence 8: --> 10:

From the original code, this is analyzed as:

flow dependence 3: --> 5:(=) (0)

which indicates a flow dependence from 3 to 5 only within the same iteration of the outer loop

```
  1: Entry
  1: integer a(1:100)
  1: integer b(1:100)
  1: integer c(1:100)
  1: integer d(1:100)
  1: integer n
  2: n = 100
  3: for i = 2,n do
  4:  a(i) = b(i)+c(i)
  5:  d(i) = a(i)
  3: endfor
```

flow dependence 2: --> 3:
flow dependence 4: --> 5:(=) (0)

```
  1: Entry
  1: integer n
  1: integer a(1:100)
  1: integer b(1:100)
  1: integer c(1:100)
  1: integer d(1:100)
  2: for i = 2,n do
  3:  a(i) = b(i)+c(i)
  4:  d(i) = a(i-1)
  2: endfor
```

flow dependence 3: --> 4:(<) (1)

This indicates thata dependency from an earlier iteration to a later iteration.

```
  1: Entry
  1: integer n
  1: integer a(1:100)
  1: integer b(1:100)
  1: integer c(1:100)
  1: integer d(1:100)
  2: for i = 2,n do
  3:  a(i) = b(i)+c(i)
  4:  d(i) = a(i+1)
  2: endfor
```

anti dependence 4: --> 3:(<) (1)

This dependence indicates that the a value being used gets overwritten in a later iteration.

```
  1: Entry
  1: integer n
  1: integer a(1:100,1:100)
  1: integer b(1:100,1:100)
  1: integer c(1:100,1:100)
  1: integer d(1:100,1:100)
  2: for i = 1,n do
  3:  for j = 2,n do
  4:   a(i,j) = a(i,j-1)+b(i,j)
  5:   c(i,j) = a(i,j)+d(i+1,j)
  6:   d(i,j) = 0
  3:  endfor
  2: endfor
```

flow dependence 4: --> 4:(=,<)  (0,1)
flow dependence 4: --> 5:(=,=)  (0,0)
anti dependence 5: --> 6:(<,=)  (1,0)

Arrays and Data Dependence

Given the ranges of loop indices and two subscripting expressions:

Might they refer to the same array entry?

Inherently pessimistic - exact analysis is time-consuming (but often feasible)

```
        do I = L,U
S1:             A(c*I + j) = . . .
S2:             . . . = A(d*I + k)
        end do
```

c, d, j, k are constants.

c*I + j =  d*I + k has a solution iff (k - j) mod GCD(c,d) = 0.

So for:

```
1: Entry
1: integer j
1: integer a(1:100)
2: for i = 1,40 do
3:  a(2*i) = i
4:  j = a(2*i+1)
2: endfor
```

The two references to A elements are different:

(1 - 0) mod GCD(2,2) = 1 ≠ 0

and there is still a dependence:

output dependence 4: --> 4:(<)  (*)

```
1: Entry
1: integer j
1: integer k
1: integer a(1:200)
2: for i = 1,10 do
3:  a(19*i+3) = j
4:  k = a(2*i+21)
2: endfor
```

(21 - 3) mod GCD(19,2) = 0

A flow dependence occurs from with I = 2 for 3: and with I = 10 for 4:

flow dependence 3: --> 4:(<=)  (*)
output dependence 4: --> 4:(<)  (*)

More powerful methods:

Integer programming (integer solution to linear program)

Omega test (Presburger arithmetic):

Natural numbers

Functions:  +, -, and multiplication by repeated addition

Predicates: $\leq, <, \geq, >, =$

Logical connectives ($\wedge, \vee, \neg$) and quantifiers ($\exists, \forall$)

Best algorithm takes $2^{2^{2^n}}$ time

Code Generation (very small sample of compiler tricks)

Loop Vectorization

```
1: Entry
1: integer a(1:100)
1: integer b(1:100)
1: integer c(1:100)
1: integer e(1:100)
2: for i = 1,99 do
3:  a(i) = b(i)
4:  c(i) = a(i)+b(i)
5:  e(i) = c(i+1)
2: endfor
```

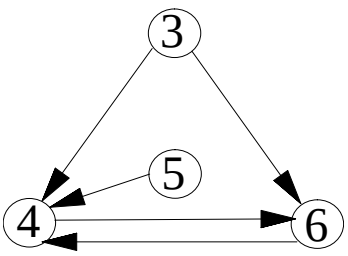
flow dependence 3: --> 4:(=) (0)
anti dependence 5: --> 4:(<) (1)

Vectorized: (Note: ''forall'' indicates that statements in the body are vectorized and each is completed before the next statement begins execution)

```
1: Entry
1: integer a(1:100)
1: integer b(1:100)
1: integer c(1:100)
1: integer e(1:100)
2: forall i = 1,99 do
3:  a(i) = b(i)
5:  e(i) = c(i+1)
4:  c(i) = a(i)+b(i)
2: endfor
```

```
1: Entry
1: integer n
1: integer a(1:100)
1: integer b(1:100)
1: integer c(1:100)
1: integer e(1:100)
2: for i = 2,n do
3:  a(i) = b(i)
4:  c(i) = a(i)+b(i-1)
5:  e(i) = c(i+1)
6:  b(i) = c(i)+2
2: endfor
```

anti dependence 3: --> 6:(=) (0)
flow dependence 3: --> 4:(=) (0)
flow dependence 4: --> 6:(=) (0)
anti dependence 5: --> 4:(<) (1)
flow dependence 6: --> 4:(<) (1)



Since 4 and 6 have a cycle of dependences, the for loop cannot be completely vectorized:

```
  1: Entry
  1: integer n
  1: integer a(1:100)
  1: integer b(1:100)
  1: integer c(1:100)
  1: integer e(1:100)
  2: forall i = 2,n do
  3:  a(i) = b(i)
  4:  e(i) = c(i+1)
  2: endfor
  6: for i = 2,n do
  7:  c(i) = a(i)+b(i-1)
  8:  b(i) = c(i)+2
  6: endfor
```

Loop Concurrentization (‘‘doall’’)

```
  1: Entry
  1: integer n
  1: integer a(1:100,1:100)
  1: integer b(1:100,1:100)
  1: integer c(1:100,1:100)
  1: integer d(1:100,1:100)
  1: integer e(1:100,1:100)
  2: for i = 1,n do
  3:  for j = 1,n do
  4:   a(i,j) = b(i,j)+c(i,j)
  5:   c(i,j) = d(i,j)/2
  6:   e(i,j) = a(i,j-1)**2+e(i,j-1)
  3:  endfor
  2: endfor
```

anti dependence 4: --> 5:(=,=)  (0,0)
flow dependence 4: --> 6:(=,<)  (0,1)
flow dependence 6: --> 6:(=,<)  (0,1)

= data dependence directions on outer loop allows it to be done in parallel:

```
  1: Entry
  1: integer n
  1: integer a(1:100,1:100)
  1: integer b(1:100,1:100)
  1: integer c(1:100,1:100)
  1: integer d(1:100,1:100)
  1: integer e(1:100,1:100)
  2: doall i = 1,n do
  3:  for j = 1,n do
  4:   a(i,j) = b(i,j)+c(i,j)
  5:   c(i,j) = d(i,j)/2
  6:   e(i,j) = a(i,j-1)**2+e(i,j-1)
  3:  endfor
  2: endfor
```

```
  1: Entry
  1: integer n
  1: integer a(1:100)
  1: integer b(1:100)
  1: integer c(1:100)
  1: integer d(1:100)
  1: integer e(1:100)
  2: for i = 2,n do
  3:  a(i) = b(i)+c(i)
  4:  c(i) = d(i)*2
  5:  e(i) = c(i)+a(i-1)
  2: endfor
```

Cannot be concurrentized due to the forward dependence:

anti dependence 3: --> 4:(=)  (0)
flow dependence 3: --> 5:(<)  (1)
flow dependence 4: --> 5:(=)  (0)

Sequent FORTRAN (precursor to OpenMP)

C$DOACROSS Compiler Directive:

1. Causes loop to be performed in parallel, assigning iterations to different processors via m_fork().
2. Has various options to ensure correct execution.
3. Does not transform in any major way; programmer does the dependency analysis.

Variable Analysis: Determine ways that the variables restrict the parallel execution.

1. Shared variables (SHARE): no restriction on loop
   a. Read-only variables
   b. Array elements are accessed (read or written) within one iteration.

   (Nested loops are executed serially.)

2. Local variables (LOCAL): use private copy instead of shared copy
   a. Temporary variables (initialized before use)
   b. Nested loop indices

   LASTLOCAL: Need value from last loop iteration

3. Reduction variable (REDUCTION): A variable used in a single var = var op expr where op commutes and associates (+, *) or can be forced to do this (-, /). Also includes MIN and MAX.

4. Shared ordered variables (SHARE): Variable (or array element) used to communicate between iterations.
   a. Must delimit by declaring region(s) where shared ordered variable is accessed

      c$order name

      c$endorder name

   b. Ensure that all iterations will access the region exactly once

   c. Ordering is achieved by: 1) spinning on a counter at beginning of ordered section and 2) setting for next iteration to continue.

5. Shared locked variables (SHARE): Variable accessed by multiple iterations, order does not matter.
   a. Often corrresponds to specialized "reduction" variables
   b. Requires locking, two ways
      1. m_lock(), m_unlock()
      2. Declare lock names in LOCKS() clause of C$DOACROSS

         C$LOCK name

         C$UNLOCK name

         Can overlap these, cannot nest.

Warshall's Algorithm for Transitive Closure

```
      for j:=1 to n do
         for i:=1 to n do
            if A[i,j] then
               for k:=1 to n do
                  A[i,k] := A[i,k] or A[j,k]
```

```
      program warshall
      dimension ia(20,20)
      do 10 j=1,n
      do 10 i=1,n
         if (ia(i,j) .eq. 1) then
            do 20 k=1,n
20             ia(i,k)=max(ia(i,k),ia(j,k))
         endif
10    continue
      stop
      end
```

DOACROSS for j-loop

| | |
|---|---|
| Shared | |
| Local | i, k |
| Reduction | |
| Shared Ordered | ia: ordered region includes entire i-loop, USELESS |
| Shared Locked | |

DOACROSS for i-loop

| | |
|---|---|
| Shared | j, ia |
| Local | k |
| Reduction | |
| Shared Ordered | |
| Shared Locked | |

DOACROSS for k-loop

| | |
|---|---|
| Shared | i, j, ia |
| Local | |
| Reduction | |
| Shared Ordered | |
| Shared Locked | |

Best is the i-loop:

```
      program warshall
      dimension ia(20,20)
      do 10 j=1,n
C$DOACROSS SHARE(j,ia),LOCAL(k)
      do 10 i=1,n
         if (ia(i,j) .eq. 1) then
            do 20 k=1,n
20             ia(i,k)=max(ia(i,k),ia(j,k))
         endif
10    continue
      stop
      end
```

Pascal's Triangle

```
        program pascal
        integer combs
        dimension combs(0:200,0:200)

        combs(0,0)=1

        do 10 i=1,200
          combs(i,0)=1
10        combs(i,i)=1

        do 20 i=2,200
          do 20 j=1,i-1
20          combs(i,j)=combs(i-1,j-1)+combs(i-1,j)
        stop
        end
```

As TINY input:

```
  1: Entry
  1: integer combs(0:200,0:200)
  2: for i = 1,200 do
  3:  combs(i,0) = 1
  4:  combs(i,i) = 1
  2: endfor
  6: for i = 2,200 do
  7:  for j = 1,i-1 do
  8:   combs(i,j) = combs(i-1,j-1)+combs(i-1,j)
  7:  endfor
  6: endfor
```

Dependences

flow dependence 3: --> 8:
flow dependence 4: --> 8:
flow dependence 8: --> 8:(<,<) (1,1)
flow dependence 8: --> 8:(<,=) (1,0)

Concurrent Version:

```
  1: Entry
  1: integer combs(0:200,0:200)
  2: doall i = 1,200 do
  3:  combs(i,0) = 1
  4:  combs(i,i) = 1
  2: endfor
  6: for i = 2,200 do
  7:  doall j = 1,i-1 do
  8:   combs(i,j) = combs(i-1,j-1)+combs(i-1,j)
  7:  endfor
  6: endfor
```

FORTRAN Version:

```
        program pascal
        integer combs
        dimension combs(0:200,0:200)

        combs(0,0)=1

c$doacross share(combs)
        do 10 i=1,200
          combs(i,0)=1
10        combs(i,i)=1

        do 20 i=2,200
c$doacross shared(combs,i)
          do 20 j=1,i-1
20          combs(i,j)=combs(i-1,j-1)+combs(i-1,j)
        stop
        end
```

1992 Lab 5: Maximize parallelism/speed for computing:

$$\sum_{i}\sum_{j}\sum_{k}\sum_{l} P_{i,k}P_{j,l}\sqrt{(i-k)^2+(j-l)^2}$$

```
      PROGRAM twod
      DIMENSION p(50,50)

      sum=0.0
      do 10 i=1,50
        do 10 j=1,50
          p(i,j)=float(i+j)
10        sum=sum+p(i,j)

      do 20 i=1,50
        do 20 j=1,50
20        p(i,j)=p(i,j)/sum

      sum=0.0
      do 30 i=1,50
       do 30 j=1,50
        do 30 k=1,50
         do 30 l=1,50
30        sum=sum+p(i,j)*p(k,l)*sqrt(float(i-k)**2+float(j-l)**2)
      print 40,sum
40    format(f8.4/)
      stop
      end
```

Solution:

```
      PROGRAM twod
      DIMENSION p(50,50),f(51),s(51,51)

      call m_set_procs(7)
      sum=0.0
c$doacross reduction(sum),local(j),share(p)
      do 10 i=1,50
        do 10 j=1,50
          p(i,j)=float(i+j)
10        sum=sum+p(i,j)

c$doacross share(p,sum),local(j)
      do 20 i=1,50
        do 20 j=1,50
20        p(i,j)=p(i,j)/sum

c$doacross share(f)
      do 25 i=1,51
25      f(i)=float(i-1)**2

c$doacross share(f,s)
      do 27 i=1,51
       do 27 j=1,51
27      s(i,j)=sqrt(f(i)+f(j))

      sum=0.0
c$doacross reduction(sum),local(j,k,l,sum2),share(p,s)
      do 30 i=1,50
       do 30 j=1,50
        sum2=0.0
        do 35 k=1,50
         do 35 l=1,50
35        sum2=sum2+p(k,l)*s(iabs(i-k)+1,iabs(j-l)+1)
30     sum=sum+p(i,j)*sum2
      print 40,sum
40    format(f8.4/)
      stop
      end
```

The ‘‘Future’’ of SMP with thread support - OpenMP

Extension of ideas in Sequent FORTRAN (common ancestry)

FORTRAN, C, and C++ APIs

Potential for tools to generate OpenMP code from sequential code

Claim: Easy to work between sequential and parallel versions by disabling compiler directives.

Claim: Pthreads is not intended for parallel programming and does not provide sufficient support.

Emphasis on work-sharing (more options than Sequent) for for-loops in legacy code. Code profiling is important.

Efforts toward mixing MPI with OpenMP.

Tutorial available from web page.