Loop-Carried Dependency Analysis and Program Transformation

Reference: Compiler transformations for High-Performance Computing. David Bacon, et al. ACM Computing Surveys, Vol. 26, No. 4.

Dependence analysis

- A *dependence* is a relationship between two computations that places constraints on their execution order. Dependence analysis identifies these constraints, which are then used to determine whether a particular transformation can be applied without changing the semantics of the computation.
- Types of dependences: (i) control dependence and (ii) data dependence.
- There is a control dependence between statement S1 and statement S2, when statement S1 determines whether S2 will be executed. For example:

```
1       if (a = 3) then
2         b = 10
        end if
```

Types of data dependence

- Flow dependence -- also called true dependence or Read after Write (RAW). Example:

  S1: a = c*10

  S2: d = 2*a + c

  Here, there is a flow dependency from S1 to S2, i.e. S1 -> S2

- Anti-dependence – also called Write after Read (WAR). Example:

  S1: e = f*4 + g

  S2: g = 2*h

Types of data dependence (continued)

- Output dependence: Both statements write to the same variable

  S1: a = b*c

  S2: a = d+e

- *Dependency Graph*: Nodes represent statements (or, blocks of statements) and edges represent dependencies between nodes

Loop carried dependency

- A simple example of loop-carried dependence is shown in the figure below. There is no dependence between S1 and S2 within any single iteration of the loop, but there is one between two successive iterations. When i = k, S2 reads the value of a[k -1] written by S1 in iteration k -1.

```
      do i = 2, n
1       a[i] = a[i] + c
2       b[i] = a[i-1] * b[i]
      end do
```

Loop carried dependency analysis

- Figure below shows a generalized perfect loop nest of d loops. The body of the loop nest reads and writes elements of the m dimensional array a. The functions fi and gi map the current values of the loop iteration variables to integers that index the i th dimension of a.

```
      do i1 = l1, u1
         do i2 = l2, u2
            ...
               do id = ld, ud
1                 a[f1(i1, ..., id), ..., fm(i1, ..., id)] = ...
2                 ... = a[g1(i1, ..., id), ..., gm(i1, ..., id)]
               end do
            ...
         end do
      end do
```

Loop carried dependency analysis (Continued)

- An iteration can be uniquely named by a vector of d elements I= (i1, ... , id), where each index falls within the iteration range of its corresponding loop in the nesting (that is, lp ≤ ip ≤ up). The outermost loop corresponds to the leftmost index.
- A reference in iteration J can depend only on another reference in iteration I that was executed before it, not after it. We formalize the notion of "before" with the << relation:

  I << J iff there exists p such that (ip <jp and for all q <p: iq =jq)

Loop carried dependency analysis (Continued)

- A reference in some iteration J depends on a reference in iteration I if and only if at least one reference is a write and

  I << J and for all p: fp(I) = gp (J)

- When I << J, we define the *dependence distance* as J - I = (j1–i1 , ... , jd-id).

- A *valid* dependence distance must be *lexicographically positive*, i.e. first non-zero entry from left to right is +ve.

Loop carried dependency analysis (Continued)

- A *dependence vector* is a collection of dependence distances.
- Example 1: For the following example, dependence vector = {(1, -1)}

```
do i = 2, n
  do j = 1, n-1
    a[i,j] = a[i,j] + a[i-1,j+1]
  end do
end do
```

Loop carried dependency analysis (Continued)

- Example 2: For the following example, dependence vector = {(1,0), (0,1), (1,-1)}

```
do i = 1, n
  do j = 2, n-1
    a[j]=(a[j] + a[j-1] + a[j+1])/3
  end do
end do
```

Loop carried dependency analysis (Continued)

- The *p th* loop in a loop nest of depth d is parallelizable if for every dependence distance V = (v1 , ... , *vp, ... , vd),*

  either *vp* = 0 or *there exits q < p such that vq* > 0.

- Example 1:

```
do   i   =   1,   n
     do j = 2, n
          a[i, j] = a[i, j-1] + c
     end do
 end do
```

Dependence vector = {(0, 1)}. Outer loop is parallelizable.

Loop carried dependency analysis (Continued)

- Example 2:

```
do i = 1, n
    do j = 1, n
        a[i, j] = a[i-1, j] + a[i-1, j+l]
    end do
end do
```

Dependence vector = {(1, 0), (1, -1)}. Inner loop is parallelizable.

Loop carried dependency analysis (Continued)

- Example 3:

  Dependence vector = {(0, 1), {1, 1)}

  None of the loop nests is parallelizable.

- Example 4:

  Dependence vector = {(0, 1), {1, 0)}

  None of the loop nests is parallelizable.

Loop reordering

- Loop interchange:

If two loops *p* and *q* in a perfect loop nest of *d* loops are interchanged, then each dependence distance $V = (v1, \ldots, vp, \ldots, vq, \ldots, vd)$ in the original loop nest becomes $V' = (v1, \ldots, vq, \ldots, vp, \ldots, vd)$ in the transformed loop nest.

If V' is lexicographically positive, then the dependence relationships of the original loop are satisfied and the interchange is legal.

Loop interchange (continued)

- Example 1:

```
do   i   =   2,   n
     do j = 1, n
          a[i, j] = a[i - 1, j] + c      //Dependence vector = {(1, 0)}. Inner loop is parallelizable.
     end do
 end do
  ↓  After interchange
do   j   =   1,   n
     do i = 2, n
          a[i, j] = a[i - 1, j] + c      //Dependence vector = {(0, 1)}. Outer loop is parallelizable.
     end do
 end do
```

Loop interchange (continued)

- Example 2:

```
do  i  =  2,  n
    do  j = 1, n - 1
        a[i, j] = a[i - 1, j + 1] + c      //Dependence vector = {(1, -1)}. Inner loop is parallelizable.
    end do
end do
```

Interchange is illegal because after interchange, dependece vector = {(-1, 1)} which is lexicographically negative.

Loop reversal

- If loop *p* in a nest of *d* loops is reversed, then for each dependence distance *V,* the entry *vp* is negated. The reversal is legal if each resulting dependence distance *V′* is lexicographically positive, that is, either *vp* = 0 or there exits q < p such that *vq* > 0.

Loop reversal (Continued)

- Example:

```
do   i   =   2,   n
     do  j = 1, n - 1
          a[i, j] = a[i - 1, j + 1] + c      //Dependence vector = {(1, -1)}.
     end do
end do
   ↓  After reversal
 do   i   =   2,   n
      do  j = n - 1, 1, -1
           a[i, j] = a[i - 1, j + 1] + c      //Dependence vector = {(1, 1)}.
      end do
end do
   ↓   Can interchange loop now
```

Cycle shrinking

- Example:

```
do i = 1, n
  1   a[i+k] = b[i]
  2   b[i+k] = a[i] + c[i]
end do
```

Because of the write to a, there is a dependency from S1 to S2 over distance k. Similarly due to the write to b, there is a dependency from S2 to S1 over distance k.

Cycle shrinking (Continued)

- Following from the previous example, the loop is transformed as follows:

```
do TI= 1, n, k
    par do i = TI, TI+k-1
       1      a[i+k] = b[i]
       2      b[i+k]  = a[i] + c[i]
    end par do
end do
```

In the above, k iterations in the inner loop can be done in parallel (i.e. *par do*).