Loop-Carried Dependences

Definition

– A dependence $D=(d_1,\ldots d_n)$ is **carried** at loop level i if d_i is the first non-zero element of D

Example

```
do i = 1,5
    do j = 2,6
        A(j,i) = B(j-1,i)+1
        B(j,i) = A(j,i-1)*2
    enddo
enddo
```

Distance vectors: (1,0) for accesses to **A**
(0,1) for accesses to **B**

Loop-carried dependences

– The i loop carries dependence due to **A**

– The j loop carries dependence due to **B**

Parallelization

Idea

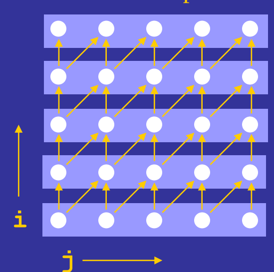
– The iterations of a loop can be executed in parallel if the loop carries no dependences

Example

```
do i = 1,5
   do j = 2,6
      A(j,i) = B(j-1,i-1)+1
      B(j,i) = A(j,i-1)*2
   enddo
enddo
```

Can we parallelize the `i` loop?

Iteration Space



Distance Vectors:
(1,0) for **A** (flow)
(1,1) for **B** (flow)

Parallelization (cont)

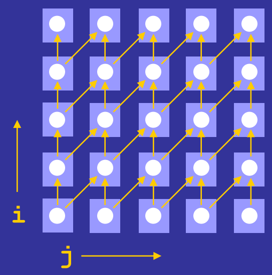
Idea

– The iterations of a loop can be executed in parallel if the loop carries no dependences

Example

```
do i = 1,5
    do j = 2,6
        A(j,i) = B(j-1,i-1)+1
        B(j,i) = A(j,i-1)*2
    enddo
enddo
```

Iteration Space



Can we instead parallelize the `j` loop?

Distance Vectors:

(1,0) for **A** (flow)
(1,1) for **B** (flow)

Scalar Expansion: Motivation

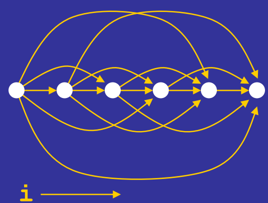
Problem

– Loop-carried dependences inhibit parallelism

– Scalar references result in loop-carried dependences

Example

```
do i = 1,6
    t = A(i) + B(i)
    C(i) = t + 1/t
enddo
```



Can this loop be parallelized? No.

What kind of dependences are these? Anti dependences.

Convention for these slides: Arrays start with upper case letters, scalars do not

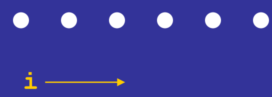
Scalar Expansion

Idea

– Eliminate false dependences by introducing extra storage

Example

```
do i = 1,6
   T(i) = A(i) + B(i)
   C(i) = T(i) + 1/T(i)
enddo
```



Can *this* loop be parallelized? Yes.

Disadvantages?

Scalar Expansion Details

Restrictions

– The loop must be a **countable** loop
  i.e. The loop trip count must be independent of the body of the loop
– There can **not** be loop-carried flow dependences due to the scalar
– The expanded scalar must have no upward exposed uses in the loop

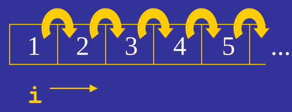
```
do i = 1,6
   print(t)
   t = A(i) + B(i)
   C(i) = t + 1/t
enddo
```

– When the scalar is live after the loop, we must move the correct array value into the scalar
– Nested loops may require much more storage

Example 2: Parallelization (reprise)

Why can't this loop be parallelized?

```
do i = 1,100
   A(i) = A(i-1)+1
enddo
```



Distance Vector: (1)

Why can this loop be parallelized?

```
do i = 1,100
   A(i) = A(i)+1
enddo
```

Distance Vector: (0)

Example 1: Loop Permutation (reprise)

Sample code

```
do j = 1,6                      do i = 1,5
   do i = 1,5                      do j = 1,6
      A(j,i) = A(j,i)+1               A(j,i) = A(j,i)+1
   enddo                           enddo
enddo                           enddo
```

Why is this legal?

– There are no loop-carried dependences, so we can arbitrarily change the order of iteration

Dependence Testing

Consider the following code

```
do i = 1,5
   A(3*i+2) = A(2*i+1)+1
enddo
```

Question

– How do we determine whether one array reference depends on another across iterations of an iteration space?

Dependence Testing in General

General code

$$
\begin{array}{l}
\texttt{do } i_1 = l_1, h_1 \\
\quad \texttt{...} \\
\quad\quad \texttt{do } i_n = l_n, h_n \\
\quad\quad\quad A(f(i_1,\ldots,i_n)) = \ldots A(g(i_1,\ldots,i_n)) \\
\quad\quad \texttt{enddo} \\
\quad \texttt{...} \\
\quad \texttt{enddo}
\end{array}
$$

There exists a dependence between iterations $I=(i_1, \ldots, i_n)$ and $J=(j_1, \ldots, j_n)$ when

– $f(I) = g(J)$

– $(l_1,\ldots l_n) < I,J < (h_1,\ldots,h_n)$

Dependence Testing: Simple Case

Sample code

```
do i = l,h
    A(a*i+c1) = ... A(a*i+c2)
enddo
```

Dependence?

$a*i_1+c_1 = a*i_2+c_2$, or

$a*i_1 - a*i_2 = c_2-c_1$

A solution exists if **a** divides c_2-c_1 evenly

Exercise

Code

i_1

```
do i = l,h
    A(2*i+2) = A(2*i-2)+1
enddo
```

i_2

Dependence?

$2*i_1 - 2*i_2 = -2 - 2 = -4$

$i_1 - i_2 = -2$ **(yes, 2 divides -4)**

Kind of dependence?

– Anti? $i_2 + d = i_1 \Rightarrow d = -2$

– Flow? $i_1 + d = i_2 \Rightarrow d = 2$

GCD Test

Idea

– Generalize test to linear functions of iterators

Code

```
do i = l_i,h_i
   do j = l_j,h_j
      A(a_1*i + a_2*j + a_0) = ... A(b_1*i + b_2*j + b_0) ...
      enddo
   enddo
```

Again

– $a_1*i_1 - b_1*i_2 + a_2*j_1 - b_2*j_2 = b_0 - a_0$
– Solution exists if $gcd(a_1,a_2,b_1,b_2)$ divides $b_0 - a_0$

Example

Code

```
do i = l_i,h_i
   do j = l_j,h_j
      A(4*i + 2*j + 1) = ... A(6*i + 2*j + 4) ...
   enddo
enddo
```

gcd(4,-6,2,-4) = 2

Does 2 divide 4-1?

Next Time

Lecture

– Loop transformations

Loop Transformations for Parallelism & Locality

Last time

– Data dependences and loops
– Loop transformations
  – Parallelization
  – Scalar expansion

Today

– Loop transformations
  – Loop reversal
  – Loop fusion
  – Loop fission
  – Loop interchange
  – Unroll and Jam

Review

Distance vectors

- Concisely represent dependences in loops (*i.e.*, in iteration spaces)
- Dictate what transformations are legal
  - *e.g.*, Permutation and parallelization

Direction vectors

- Compare $\mathbf{i}^S$ and $\mathbf{i}^T$: <, >, =

Legality

- A dependence vector is **legal** when it is lexicographically nonnegative

Loop-carried dependence

- A dependence $D=(d_1,...d_n)$ is **carried** at loop level i if d_i is the first nonzero element of D

Loop Reversal

Idea

- Change the direction of loop iteration
  (*i.e.*, From low-to-high indices to high-to-low indices or vice versa)

Benefits?

- Improved cache performance
- Enables other transformations (coming soon)

Example

```
do i = 6,1,-1
   A(i) = B(i) + C(i)
enddo
```

→

```
do i = 1,6
   A(i) = B(i) + C(i)
enddo
```

Loop Reversal and Distance Vectors

Impact

- Reversal of loop *i* negates the i^{th} entry of all distance vectors associated with the loop
- What does it do to direction vectors?

When is reversal legal?

- When the loop being reversed does not carry a dependence

(*i.e.*, When the transformed distance vectors remain legal)

Example

```
do i = 1,5
   do j = 1,6
      A(j,i) = A(j-1,i-1)+1
   enddo
enddo
```

Dependence: Flow
Distance Vector: (1,1)

Can either loop be reversed?

April 29, 2015 Loop Transformations 19

Loop Reversal Example

Legality

- Loop reversal will change the direction of the dependence relation

Is the following legal?

```
do i = 1,6
   A(i) = A(i-1)
enddo
```

Dependence: Flow
Distance Vector: (1)

```
do i = 6,1,-1
   A(i) = A(i-1)
enddo
```

Dependence: Anti
Distance Vector: (1)

April 29, 2015 Loop Transformations 20

Loop Fusion

Idea

– Combine multiple loop nests into one

Example

```
do i = 1,n
   A(i) = A(i-1)
enddo
do j = 1,n
   B(j) = A(j)/2
enddo
```

→

```
do i = 1,n
   A(i) = A(i-1)
   B(i) = A(i)/2
enddo
```

Pros?

– May improve data locality
– Reduces loop overhead
– May enable better instruction scheduling
– Enables **array contraction** (opposite of scalar expansion)

Cons?

– May hurt i-cache performance
– May hurt data locality How?

Loop Fusion Can Hurt Locality

Example

```
do j = 1,n                          reuse
   do i = 1,m
      B(i,j) = A(i,j) + A(i,j-1) + A(i,j-2)
   enddo
enddo           reuse
do j = 1,n
   do i = 1,m
      C(i,j) = B(i,j) + D(i,j)
   enddo
enddo
```

Loop Fusion Can Hurt Locality (cont)

After fusion

```
do j = 1,n
   do i = 1,m
      B(i,j) = A(i,j) + A(i,j-1) + A(i,j-2)
      C(i,j) = B(i,j) + D(i,j)
   enddo
enddo
```

Lost reuse

Saved loads

Legality of Loop Fusion

Basic Requirements

– Both loops must have same structure
  – Same loop depth
  – Same loop bounds
  – Same iteration directions

Can we relax any of these restrictions?

– Dependences must be preserved
  e.g., Flow dependences must not become anti dependences

```
do i = 1,n
   body1
enddo
do i = 1,n
   body2
enddo
```

All cross-loop dependences flow from body1 to body2

```
do i = 1,n
   body1
   body2
enddo
```

Ensure that fusion does not introduce dependences from body2 to body1

Loop Fusion Example

What are the dependences?

```
    do i = 1,n
s1      A(i) = B(i) + 1
    enddo
    do i = 1,n
s2      C(i) = A(i)/2
    enddo
    do i = 1,n
s3      D(i) = 1/C(i+1)
    enddo
```

$s_1\delta^f s_2$

$s_2\delta^f s_3$

What are the dependences?

```
    do i = 1,n
s1      A(i) = B(i) + 1
s2      C(i) = A(i)/2
s3      D(i) = 1/C(i+1)
    enddo
```

$s_1\delta^f s_2$

$s_3\delta^a s_2$

Fusion changes the dependence between s_2 and s_3, so fusion is illegal

Is there some transformation that will enable fusion of these loops?

Loop Fusion Example (cont)

Loop reversal is legal for the original loops

- Does not change the direction of any dependence in the original code
- Will reverse the direction in the fused loop: $s_3\delta^a s_2$ will become $s_2\delta^f s_3$

```
    do i = n,1
s1      A(i) = B(i) + 1
    enddo
    do i = n,1
s2      C(i) = A(i)/2
    enddo
    do i = n,1
s3      D(i) = 1/C(i+1)
    enddo
```

$s_1\delta^f s_2$

$s_2\delta^f s_3$

```
    do i = n,1
s1      A(i) = B(i) + 1
s2      C(i) = A(i)/2
s3      D(i) = 1/C(i+1)
    enddo
```

$s_1\delta^f s_2$

$s_2\delta^f s_3$

After reversal and fusion all original dependences are preserved

Loop Fission (Loop Distribution)

Idea

- Split a loop nest into multiple loop nests (the inverse of fusion)

Example

```
do i = 1,n
   A(i) = B(i) + 1
   C(i) = A(i)/2
enddo
```

⇒

```
do i = 1,n
   A(i) = B(i) + 1
enddo

do i = 1,n
   C(i) = A(i)/2
enddo
```

Motivation?

- Produces multiple (potentially) less constrained loops
- May improve locality
- Enable other transformations, such as interchange

Legality?

Loop Fission (cont)

Legality

- Fission is legal when the loop body contains no cycles in the dependence graph

```
do i = 1,n
   body1
   body2
enddo
```

⇒

```
do i = 1,n
   body1
enddo
do i = 1,n
   body2
enddo
```

Cycles cannot be preserved because after fission all cross-loop dependences flow from body1 to body2