Class Announcements

Here is where we are:

- Second project due Monday, April 24
- Third project has been posted; due Monday, May 1
- Sixth homework due Tuesday, April 25
- Seventh homework will be posted next week; due Monday, May 1
- Final (third midterm) exam on Thursday, May 4, noon - 3:00pm timeslot. Location: most likely this room.

  Any CONFLICTS with other classes?

Review - Dependence Testing

Given

```
 do i_1 = L_1, U_1
    ...
       do i_n = L_n, U_n
S_1          A(f_1(i_1,...,i_n),...,f_m(i_1,...,i_n))   =   ...
S_2          ...   =   A(g_1(i_1,...,i_n),...,g_m(i_1,...,i_n))
```

A *dependence* between statement S_1 and S_2, denoted $S_1 \delta S_2$, indicates that S_1, the *source*, must be executed before S_2, the *sink* on some iteration of the nest.

Let α & β be a vector of n integers within the ranges of the lower and upper bounds of the n loops.

Does $\exists\ \alpha \leq \beta$, s.t.

$$f_k(\alpha) \ = \ g_k(\beta) \quad \forall k,\ 1 \leq k \leq m\ ?$$

Approaches to Dependence Testing

- can we solve this problem exactly?
- what is conservative in this framework?
- restrict the problem to consider index and bound expressions that are linear functions

$\Longrightarrow$ solving general system of linear equations in integers is NP-hard

Solution Methods

- inexact methods
  - Greatest Common Divisor (GCD)
  - Banerjee's inequalities
- cascade of exact, efficient tests (fall back on inexact methods if needed)
  - Rice (see posted PLDI'91 paper)
  - Stanford
- exact general tests (integer programming)

Dependence Testing

SIV - Single Induction Variable Test

1. Single loop nest with constant lower (LB) and upper (UB) bounds, and step 1

```
for i = LB, UB, 1
 ...
endfor
```

   The loop bounds define the iteration space for loop induction variable `i`.

2. Two array references with array subscript (index) expressions of the form (true dependence)

```
   for i = LB, UB, 1
R1:   X(a*i + c1) = ...       \\ write
R2:      ... X(a*i + c2) ...  \\ read
   endfor
```

   where `a`, `c1`, and `c2` are integer constants, R1 and R2 are references to the same array, `i` is the loop induction variable, and `a` $\neq 0$.

Question: $\boxed{\text{Is there a true dependence between R1 and R2?}}$

Dependence Testing

There is a dependence between R1 and R2 **iff**

$$\exists i, i' : i \leq i' \;\; and \;\; (a * i + c_1) = (a * i' + c_2)$$

where i and i' are two iterations in the iteration space of the loop. This means that in both iterations, the same element of array `X` would be accessed.

So let's just solve the equation:

$$(a * i + c_1) = (a * i' + c_2) \Leftrightarrow$$

$$\frac{c_1 - c_2}{a} = i' - i = \Delta d$$

There is a dependence with distance Δd iff

1. Δd is an integer value and
2. UB - LB $\geq \Delta d \geq 0$

Dependence Testing Examples

1.
```
       for i = LB, UB, 1
   R1:   X(i) = ...            \\ write
   R2:      ... X(i - 2) ...  \\ read
       endfor
```

a=1, c_1=0, c_2=-2 $\Rightarrow \Delta d = 2$ (**dependence**)

2.
```
       for i = LB, UB, 1
   R1:   X(2*i) = ...            \\ write
   R2:      ... X(2*i - 1) ...  \\ read
       endfor
```

a=2, c_1=0, c_2=-1 $\Rightarrow \Delta d = \frac{1}{2}$ (**no dependence**)

Assume R1 executes before R2.

$\boxed{\text{Classification of dependences}}$:

- R1 is write, R2 is read $\Rightarrow$ **true** dependence
- R1 is read, R2 is write $\Rightarrow$ **anti** dependence
- R1 is write, R2 is write $\Rightarrow$ **output** dependence

Dependence Testing

ZIV - Zero Induction Variable Test

Two array references with array subscript (index) expressions of the form of a constant:

```
   for i = LB, UB, 1
R1:   X(c1) = ...        \\ write
R2:      ... X(c2) ...   \\ read
   endfor
```

where `c1`, and `c2` are integer constants, and R1 and R2 are references to the same array.

There is a dependence between R1 and R2 **iff**

$$c_1 = c_2 = c.$$

What is the dependence distance Δd?

Since every iteration i writes `X(c)`, and every iteration i' reads `X(c)`, there is no fixed distance Δd. In fact, both references have true, anti, and output dependences:

$$\Delta d \in \{0, \ldots UB - LB\} \text{ for true}$$

$$\Delta d \in \{1, \ldots UB - LB\} \text{ for anti and output}$$

Project and OpenMP

Two important issues while specifying the parallel execution of a `for` loops:

- **safety** – parallel execution has to preserve all dependences
- **profitability** – benefits of parallel execution have to compensate for the overhead penalty

Project and OpenMP

safety

Sample code:

```
#pragma omp parallel for private(i, hash)
     for (j = 0; j < num_hf; j++) {
          for (i = 0; i < wl_size; i++) {
               hash = hf[j] (get_word(wl, i));
               hash %= bv_size;
               bv[hash] = 1;
          }
     }
```

This specifies:

- outermost (`j`-loop) is parallel
- each thread will get its own copy of variables `i` and `hash`, eliminating loop carried anti and output dependences.

profitability

Sample code:

```
#define CHUNK_SIZE 2
int chunk = CHUNK_SIZE
#pragma omp parallel for \
        schedule (dynamic, chunk) \
        private(i, hash)
     for (j = 0; j < num_hf; j++) {
          for (i = 0; i < wl_size; i++) {
               hash = hf[j] (get_word(wl, i));
               hash %= bv_size;
               bv[hash] = 1; } }
```

'

This specifies:

- outermost (`j`-loop) is parallel, with CHUNK_SIZE iterations scheduled as a group; default chunk size=1
- three basic scheduling strategies:
  static, **dynamic**, or **guided**

There are many more options of specifying how to execute `for` loops in parallel (see the online OpenMP tutorial)

Loop Transformations

Goal

- modify execution order of loop iterations
- preserve data dependence constraints

Motivation

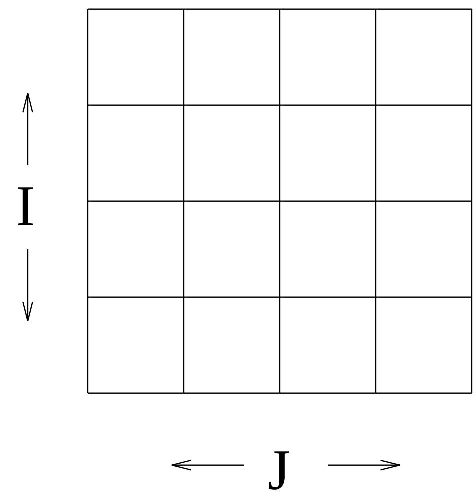
- data locality
  (increase reuse of registers, cache)
- parallelism
  (eliminate loop-carried deps, incr granularity)

Taxonomy

- loop interchange
  (change order of loops in nest)
- loop fusion
  (merge bodies of adjacent loops)
- loop distribution
  (split body of loop into adjacent loops)

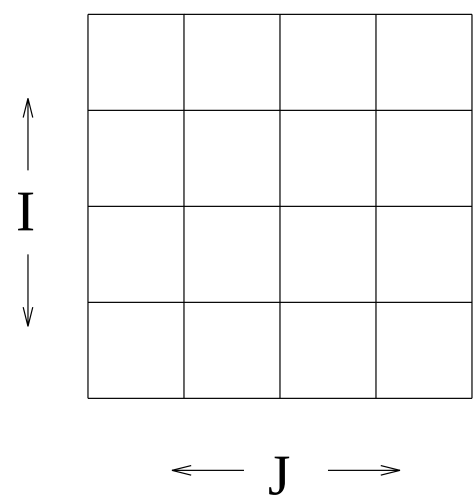
Loop Interchange

```
   do I = 1, N
     do J = 1, N
S1     A(I,J) = A(I,J-1)
S2     B(I,J) = B(I-1,J-1)
     enddo
   enddo
```



$\Longrightarrow$ loop interchange $\Longrightarrow$

```
   do J = 1, N
     do I = 1, N
S1     A(I,J) = A(I,J-1)
S2     B(I,J) = B(I-1,J-1)
     enddo
   enddo
```



Loop interchange is safe *iff*

- it does not create a lexicographically negative direction vector (1,–1) $\rightarrow$ (–1,1)

$\Rightarrow$ Benefits

- may expose parallel loops, incr granularity
- reordering iterations may improve reuse

Loop Fusion

```
      do i = 2, N
S1      A(i) = B(i)
      do i = 2, N
S2      B(i) = A(i-1)
```

$\Longrightarrow$ loop fusion $\Longrightarrow$

```
      do i = 2, N
S1      A(i) = B(i)
S2      B(i) = A(i-1)
```

Loop fusion is safe *iff*

- no loop-independent dependence between nests is converted to a backward loop-carried dep

  (would fusion be safe if S_2 referenced `a(i+1)` ?)

$\Rightarrow$ Benefits

- reduces loop overhead
- improves reuse between loop nests
- increases granularity of parallel loop

Loop Distribution (Fission)

```
     do i = 2, N
S1     A(i) = B(i)
S2     B(i) = A(i-1)
```

$\Longrightarrow$ loop distribution $\Longrightarrow$

```
     do i = 2, N
S1     A(i) = B(i)

     do i = 2, N
S2     B(i) = A(i-1)
```

Loop distribution is safe *iff*

- statements involved in a cycle of true deps (*recurrence*) remain in the same loop, and
- if $\exists$ a dependence between two statements placed in different loops, it must be forward

$\Rightarrow$ Benefits

- necessary for vectorization
- may enable partial/full parallelization
- may enable other loop transformations
- may reduce register/cache pressure

A Simple Vectorizing Compiler

How to vectorize the following loops?

```
for (i=2; i<100; i++) {
  S1:  a[i] = b[i+1] + 1;
  S2:  b[i] = a[i] + 5;
}
```

```
for (i=2; i<100; i++) {
  S1:  a[i] = b[i-1] + a[i-1] + 3;
  S2:  b[i] = a[i+1] + 5;
}
```

Simple vectorizer assumptions:

1. singly-nested loops
2. constant upper and lower bounds, step is always 1
3. body is sequence of assignment statements to array variables
4. simple array index expressions of induction variable (i +/- c or c); can use ZIV or SIV test
5. no function calls

A Simple Vectorizing Source-to-Source Compiler

SKETCH OF BASIC ALGORITHM

Here is a basic vectorization algorithm based on a statement-level dependence graph:

1. Construct statement-level dependence graph considering true, anti, and output dependences; in the final dependence graph, the type of the dependence is not important any more
2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do
   (a) if SCC has more than one statement in it, distribute loop with statements of SCC as its body, and keep the code sequential
   (b) if SCC is a single statement and has no dependence cycle, distribute loop around it and generate vector code; otherwise, mark distributed loop sequential.

Next Lecture

Things to do:

- Quantum computing basics
- OpenMP tutorial on our website