Class Information

- My office hours are moved to Thursday, May 1, 10:00am - noon.
- Last, non-graded homework has been posted together with its sample solution.
- Last lecture: Friday, May 2.
- Final exam: May 8, noon to 3:00pm. Location will be posted on our web site later.
- Possible review session on Tuesday, May 6.
- Note: "Final grades" reported through our sakai grading system are only an approximation and not your official final grade.

Review - Loop Transformations

Goal

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

Motivation

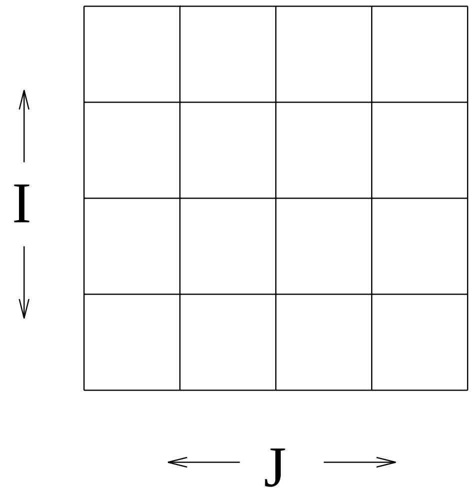
- data locality
  (increase reuse of registers, cache)
- parallelism
  (eliminate loop-carried deps, incr granularity)

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)

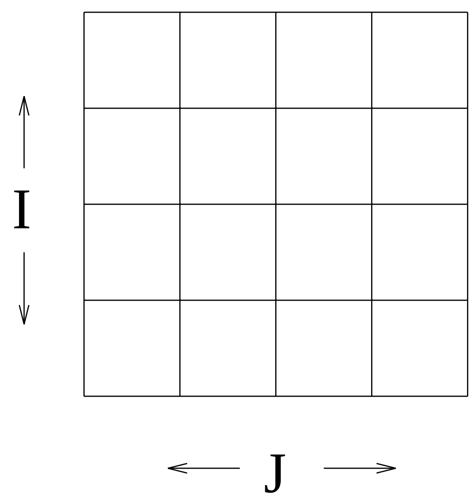
Loop Interchange

```
      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 dependencies (*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 Vectorizing Source-to-Source Compiler

EXAMPLE

```
for (i=2; i<99; i++) {
  S1:  a[i] = b[i-1] + c[i-1] + 3;
  S2:  b[i] = (c[i] + b[i+1]) / 2;
  S3:  c[i] = a[i] + 1;
  S4:  d[i] = b[i] + c[i+1];
}
```

A Vectorizing Source-to-Source Compiler

EXAMPLE

```
S2: b[2:99] = (c[2:99] + b[3:100]) / 2;

S4: d[2:99] = b[2:99] + c[3:100];

for (i=2; i<99; i++) {
   S1:  a[i] = b[i-1] + c[i-1] + 3;
   S3:  c[i] = a[i] + 1;
}
```

Type Errors and Type Systems

Scott, Chapter 7.2; ALSU Chapter 6.5

Type Error: Applying a function of type $S \rightarrow T$ to an argument not of type S

Goal: No type error remains undetected, i.e., type errors are detected before they actually occur.

How to achieve this goal?

Each language construct (operator, expression, statement, ...) has a type.

basic types: integer, real, character, ...

constructed types: arrays, records, sets, pointers, functions

A type system is a collection of *rules* for assigning *type expressions* to operators, expressions, ... in the program. Type systems are language dependent.

A type checker implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system. The type checker "computes" or "reconstructs" type expressions.

Type expressions

1. A basic type is a type expression. A special basic type, *typeError* will signal an error. A basic type *void* denotes an untyped statement.

2. Since type expressions may be named, a type name is a type expression. (e.g.: `typedef struct foo bar;`)

3. Type expressions may contain variables whose values are type expressions (e.g.: useful for languages without type declarations, or polymorphism).

4. A *type constructor* applied to type expressions is a type expression. Examples:
   (a) arrays
   (b) cartesian products
   (c) records
   (d) pointers
   (e) functions

Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer (Pascal definition).
  Rule for $+$ (analogue rules for $-$ and $*$):

$$\frac{E \vdash e_1 : integer \quad E \vdash e_2 : integer}{E \vdash (e_1 + e_2) : integer}$$

where E is a *type environment* that maps constants and variables to their types.

In combination with the following two axioms in the type system $\{c : \alpha\} \vdash c : \alpha$ we can now infer, that $(2 + 3)$ is of type integer:

$$\frac{E \vdash 2 : integer \quad E \vdash 3 : integer}{E \vdash (2 + 3) : integer}$$

where $E = \{2 : integer, 3 : integer\}$.

In general, type deduction proofs work bottom up.