

## Fine-Grained and Coarse-Grained Parallelism

Chapter 5 & 6 of Allen & Kennedy: *Optimizing Compilers for Modern Architectures*

CSC 2/455  
Fall 2004

Lecturer: Yutao Zhong

## Overview

- ❖ Theme: the use of dependence to automatically parallelize sequential code
- ❖ Outline
  - ❖ Fine-grained parallelism: vectorization
    - loop interchange
    - scalar expansion
    - scalar and array renaming
    - node splitting
  - ❖ Coarse-grained parallelism: multiple asynchronous processor
    - loop interchange

URCS

CS2/455 2004

2

## Review for *codegen* (Ch2)

```

procedure codegen(R, k, D);
// R is the region for which we must generate code.
// k is the minimum nesting level of possible parallel loops.
// D is the dependence graph among statements in R.
find the set {S1, S2, ..., Sm} of maximal strongly-connected
regions in the dependence graph D restricted to R
construct Rpi from R by reducing each Si to a single node and
compute Dpi, the dependence graph naturally induced on Rpi by D
let {pi1, pi2, ..., pim} be the m nodes of Rpi numbered in an order
consistent with Dpi (use topological sort to do the numbering);
for i = 1 to m do begin
    if pii is cyclic then begin
        generate a level-k DO statement;
        let Di be the dependence graph consisting of all dependence edges in D
        that are at level k+1 or greater and are internal to pii;
        codegen(pii, k+1, Di);
        generate the level-k ENDDO statement;
    end
    else
        generate a vector statement for pii in r(pi)-k+1 dimensions, where r(pi) is
        the number of loops containing pii;
    end
end
    
```

URCS

CS2/455 2004

3

## Review of *codegen* (Ch2)

- ❖ Basic idea: find all the possible parallelism by loop distribution and statement reordering
- ❖ May not work as desired
  - ❖ not effectively
    - loop interchange to increase parallelism
  - ❖ cyclic dependences exist
    - scalar expansion
    - scalar & array renaming
    - node splitting

URCS

CS2/455 2004

4

## Loop interchange

```

DO I = 1, N
  DO J = 1, M
    A(I, J+1) = A(I, J) + B
  ENDDO
ENDDO
    
```

$DV(I, J) = (\neq, <)$

- ❖ After interchanging I-loop and J-loop

```

DO J = 1, M
  DO I = 1, N
    A(I, J+1) = A(I, J) + B
  ENDDO
ENDDO
    
```

$DV(J, I) = (<, \neq)$

- ❖ Vectorization

```

DO J = 1, M
S  A(1:N, J+1) = A(1:N, J) + B
ENDDO
    
```

URCS

CS2/455 2004

5

## Loop interchange: safety

- ❖ Loop interchange is a reordering transformation
- ❖ Not all loop interchanges are legal (Theorem 2.3)

```

DO J = 1, M
  DO I = 1, N
    A(I, J+1) = A(I+1, J) + B
  ENDDO
ENDDO
    
```

$DV(J, I) = (<, >)$

- ❖ Theorem 5.1 : let D be a direction vector for a dependence in a perfect loop nests, the direction vector of the same dependence after any loop permutation is determined by applying the same permutation to the elements of D

URCS

CS2/455 2004

6

## Loop interchange: safety

```

DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J+1,K) = A(I,J,K) + A(I,J+1,K+1)
    ENDDO
  ENDDO
ENDDO

```

$$\begin{pmatrix} < < = \\ < = > \end{pmatrix}$$

- ❖ The *direction matrix* for a nest of loops is a matrix in which each row is a direction vector for some dependence between statements contained in the nest and every such direction vector is represented by a row.

URCS

CS2/455 2004

7

## Loop interchange: safety

- ❖ Theorem 5.2: A permutation of the loops in a perfect nest is legal if and only if the direction matrix, after the same permutation is applied to its columns, has no ">" direction as the leftmost non-"=" direction in any row.

$$\begin{matrix} I & J & K \\ \begin{pmatrix} < < = \\ < = > \end{pmatrix} \end{matrix}$$

- ❖ Interchange preventing dependence and interchange sensitive dependence

URCS

CS2/455 2004

8

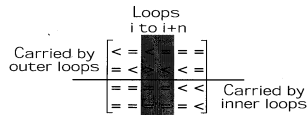
## Loop interchange for vectorization

- ❖ Motivation

- ❖ If we deal with loops containing cyclic dependences early on in the loop nest, we can potentially vectorize more loops

- ❖ Inward-shifting loops that carry no dependences

- ❖ Theorem 5.3: In a perfect loop nest, loops carry no dependence are legal to be shifted inward and will not carry any dependences in their new position.



URCS

CS2/455 2004

9

## Loop shifting: example (I)

```

DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      S A(I,J) = A(I,J) + B(I,K)*C(K,J)
    ENDDO
  ENDDO
ENDDO

```

- ❖ Applying *codegen*: no vectorization possible
- ❖ DV(I,J,K) = (=,=,<)

URCS

CS2/455 2004

10

## Loop shifting: example (II)

- ❖ After shifting K-loop outward:

```

DO K = 1, N
  DO I = 1, N
    DO J = 1, N
      S A(I,J) = A(I,J) + B(I,K)*C(K,J)
    ENDDO
  ENDDO
ENDDO

```

- ❖ Applying codegen:

```

DO K = 1, N
  FORALL J=1,N
    A(1:N,J) = A(1:N,J) + B(1:N,K)*C(K,J)
  END FORALL
ENDDO

```

URCS

CS2/455 2004

11

## Loop interchange: profitability

- ❖ The architecture of the target machine is usually the principal factor:

- ☛ SIMD
  - ☛ Vector register machine
  - ☛ MIMD
    - will see more in Chapter 6

URCS

CS2/455 2004

12

## Codegen revised

```

if pi is cyclic then
  if k is the deepest loop in pi,
    then try_recurrence_breaking(pi, D, k)
  else begin
    select_loop_and_interchange(pi, D, k);
    generate a level-k DO statement;
    let D be the dependence graph consisting of
    all dependence edges in D that are at level
    k+1 or greater and are internal to pi;
    codegen(pi, k+1, D);
    generate the level-k ENDDO statement
  end
end
end

```

{ scalar expansion  
 scalar renaming  
 array renaming  
 node splitting

URCS

CS2/455 2004

13

## Scalar expansion

```

DO I = 1, N
  S1 T = A(I)
  S2 A(I) = B(I)
  S3 B(I) = T
ENDDO

```

❖ Scalar expansion:

```

DO I = 1, N
  S1 T$(I) = A(I)
  S2 A(I) = B(I)
  S3 B(I) = T$(I)
ENDDO
T = T$(N)

```

❖ Vectorization:

```

S1 T$(1:N) = A(1:N)
S2 A(1:N) = B(1:N)
S3 B(1:N) = T$(1:N)
T = T$(N)

```

URCS

CS2/455 2004

14

## Scalar expansion: safety & profitability

- ❖ Always safe to be applied
- ❖ Not always profitable

```

DO I = 1, N
  T = T + A(I) + A(I-1)
  A(I) = T
ENDDO

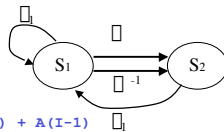
```

after scalar expansion:

```

T$(0) = T
DO I = 1, N
  S1 T$(I) = T$(I-1) + A(I) + A(I-1)
  S2 A(I) = T$(I)
ENDDO
T = T$(N)

```



URCS

CS2/455 2004

15

## Scalar expansion: profitability

- ❖ Dependences due to reuse of memory location vs. reuse of values
  - ☞ Dependences due to reuse of values must be preserved
  - ☞ Dependences due to reuse of memory location can be deleted by expansion
- ❖ Naïve approach
- ❖ Better approach: detecting deletable edges first, then applying scalar expansion only if the new graph has vectorizable statements
  - ☞ covering definition

URCS

CS2/455 2004

16

## Scalar expansion: drawbacks

- ❖ Increased memory assumption
- ❖ Solutions:
  - ☞ Expand in a single loop
  - ☞ Strip mine loop before expansion
  - ☞ Forward substitution:

```

DO I = 1, N
  T = A(I) + A(I+1)
  A(I) = T + B(I)
ENDDO

DO I = 1, N
  A(I) = A(I) + A(I+1) + B(I)
ENDDO

```

URCS

CS2/455 2004

17

## Scalar renaming: example

```

DO I = 1, 100
  S1 T = A(I) + B(I)
  S2 C(I) = T + T
  S3 T = D(I) - B(I)
  S4 A(I+1) = T * T
ENDDO

```

After renaming T:

```

DO I = 1, 100
  S1 T1 = A(I) + B(I)
  S2 C(I) = T1 + T1
  S3 T2 = D(I) - B(I)
  S4 A(I+1) = T2 * T2
ENDDO

```

❖ Vectorization:

```

S3 T2$(1:100) = D(1:100) - B(1:100)
S4 A(2:101) = T2$(1:100) * T2$(1:100)
S1 T1$(1:100) = A(1:100) + B(1:100)
S2 C(1:100) = T1$(1:100) + T1$(1:100)
T = T2$(100)

```

URCS

CS2/455 2004

18

## Scalar renaming

- Renaming algorithm partitions all definitions and uses of a scalar  $S$  into equivalent classes, each of which can occupy a different memory location (Fig 5.12)

- def-use graph
- reachable analysis

- Renaming works by removing *critical* loop-independent anti- or output- dependence to break a cycle

URCS

CS2/455 2004

19

## Array renaming

- Original:

```
DO I = 1, N
S1  A(I) = A(I-1) + X
S2  Y(I) = A(I) + Z
S3  A(I) = B(I) + C
ENDDO
```

- After renaming A:

```
DO I = 1, N
S1  A$(I) = A(I-1) + X
S2  Y(I) = A$(I) + Z
S3  A(I) = B(I) + C
ENDDO
```

- Vectorization:

```
S3  A(1:N) = B(1:N) + C
S1  A$(1:N) = A(0:N-1) + X
S2  Y(1:N) = A$(1:N) + Z
```

URCS

CS2/455 2004

20

## Node splitting

```
DO I = 1, N
S1  A(I) = X(I+1) + X(I)
S2  X(I+1) = B(I) + 10
ENDDO
```

- renaming does not work because of the two dependences share one single access:  $X(I+1)$
- renaming tries to give both name spaces the original array name
- solution: creating a copy of a node from which the critical anti-dependence emanates

URCS

CS2/455 2004

21

## Node splitting: example

- Original:

```
DO I = 1, N
S1  A(I) = X(I+1) + X(I)
S2  X(I+1) = B(I) + 10
ENDDO
```

- After node splitting:

```
DO I = 1, N
S1'  X$(I) = X(I+1)
S1  A(I) = X$(I) + X(I)
S2  X(I+1) = B(I) + 10
ENDDO
```

- Vectorization:

```
S1'  X$(1:N) = X(2:N+1)
S2  X(2:N+1) = B(1:N) + 10
S1  A(1:N) = X$(1:N) + X(1:N)
```

URCS

CS2/455 2004

22

## Node splitting: profitability

- Not always profitable

- For example

```
DO I = 1, N
S1  A(I) = X(I+1) + X(I)
S2  X(I+1) = A(I) + 10
ENDDO
```

- After splitting as before:

```
DO I = 1, N
S1'  X$(I) = X(I+1)
S1  A(I) = X$(I) + X(I)
S2  X(I+1) = A(I) + 10
ENDDO
```

- Recurrence not broken

- Why?

The target dependence must be critical!

URCS

CS2/455 2004

23

## Fine-grained parallelism: summary

```
if pi is cyclic then
  if k is the deepest loop in pi
    then try_recurrence_breaking(pi, D, k)
  else begin
    select_loop_and_interchange(pi, D, k);
    generate a level-k DO statement;
    let D, be the dependence graph consisting of
    all dependence edges in D that are at level
    k+1 or greater and are internal to pi;
    codegen(pi, k+1, D);
    generate the level-k ENDDO statement
  end
end
```

scalar expansion  
scalar renaming  
array renaming  
node splitting

URCS

CS2/455 2004

24

## Coarse-grained parallelism

- ❖ Target machine: symmetric multiprocessor
  - ✦ multiple processors with a shared memory
  - ✦ parallelism employed by creating and executing a process on each processor
  - ✦ expensive overhead: processes initiation and synchronization
- ❖ Parallelism concern for high performance:
  - ✦ find and package parallelism with a granularity large enough to compensate the overhead
  - ✦ delicate trade-off between overhead minimization and load balancing

URCS

CS2/455 2004

25

## Basics

- ❖ Sequential loop vs. parallel loop
  - ✦ a sequential loop carries a dependence
  - ✦ iterations of a parallel loop can be correctly run in any order
- ❖ Statement PARALLEL DO
  - ✦ represents a parallel loop that can be distributed on different processors
- ❖ Synchronization
  - ✦ barrier: forces all processes to reach a certain point before execution continues

URCS

CS2/455 2004

26

## Loop interchange: revisited

- ❖ Previously: fine-grained
  - ✦ move loops inward to vectorize more
- ❖ Now: coarse-grained
  - ✦ move dependence-free loops outward to generate large enough parallel unit

```
DO I = 1, N
  DO J = 1, M
    A(I+1, J) = A(I, J) + B(I, J)
  ENDDO
ENDDO
```

$DV(I,J) = (<, =)$   
**N barriers needed**

URCS

CS2/455 2004

27

## Loop interchange: example

- ❖ After interchanging I-loop and J-loop:

```
PARALLEL DO J = 1, N
  DO I = 1, N
    A(I+1, J) = A(I, J) + B(I, J)
  ENDDO
END PARALLEL DO
```

$DV(J,I) = (=, <)$   
**1 barrier needed**

URCS

CS2/455 2004

28

## Loop interchange: profitability

- ❖ Not always possible to move a parallel loop outward and have it remain free of dependence

- ❖ Example:

```
DO J = 1, N
  DO I = 1, N
    A(I+1, J+1) = A(I, J) + B(I, J)
  ENDDO
ENDDO
```

$DV(J,I) = (<, <)$

- ✦ The best we can do: parallelize the inner loop

URCS

CS2/455 2004

29

## Loop interchange: profitability

- ❖ Theorem 6.3: In a perfect nest of loops, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only “=” entries.

URCS

CS2/455 2004

30

## Loop interchange: algorithm

- ❖ Move any loop with only “=” entries into outermost position and parallelize it, remove the column from the matrix
- ❖ Move any loop with the most “<” entries into next outermost position and sequentialize it, eliminate the column and any rows representing carried dependences
- ❖ Repeat the algorithm starting with step 1

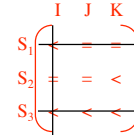
URCS

CS2/455 2004

31

## Loop selection and interchange

```
DO I=1,N
  DO J=1,M
    DO K=1,L
      S1   A(I+1,J,K) = A(I,J,K) + X1
      S2   B(I,J,K+1) = B(I,J,K) + X2
      S3   C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  ENDDO
ENDDO
```



URCS

CS2/455 2004

32

## Loop selection and interchange

```
DO I=1,N
  PARALLEL DO J=1,M
    DO K=1,L
      S1   A(I+1,J,K) = A(I,J,K) + X1
      S2   B(I,J,K+1) = B(I,J,K) + X2
      S3   C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  END PARALLEL DO
ENDDO
```

❖ general case: NP-complete

URCS

CS2/455 2004

33

## Summary

- ❖ Transformations to break recurrence
  - ❖ scalar expansion
  - ❖ scalar/array renaming
  - ❖ node splitting
- ❖ Transformations to increase parallelism
  - ❖ loop interchange
    - fine-grained: vectorization
    - coarse-grained: multiprocessor

URCS

CS2/455 2004

34