CSC 255/455

Dependence Theory and High-level Program Transformations

Instructor: Chen Ding

Spring 2014

1K x 1K Matrix Multiply

- Machine and compiler (tested around 2003)
  - SGI, MIPS R12K 250MHz, MIPSpro compiler
  - Intel, Intel Pentium4 2GHz, *GCC* compiler
  - IBM, Power4 1GHz, Xlc compiler
  - Sun, Ultra5 360MHz, Sun compiler

	Intel 2GHz	IBM 1GHz	Sun 360MHz	SGI 250MHz
no opt				
scalar opt				
loop opt				

Dependence Theory and Practice

What we will cover

- Introduction to Dependences
- Loop-carried and Loop-independent Dependences
- Allen-Kennedy vectorization algorithm
- Loop-nest transformations

Dependences

- We will concentrate on data dependences
- Chapter 7 deals with control dependences
- Simple example of data dependence:

```
S1  PI = 3.14
S2  R = 5.0
S3  AREA = PI * R ** 2
```

- Statement S_3 cannot be moved before either S_1 or S_2 without compromising correct results

Load Store Classification

- Quick review of dependences classified in terms of load-store order:
  1. True dependences (RAW hazard)
     - S_2 depends on S_1 is denoted by $S_1 \delta S_2$
  2. Antidependence (WAR hazard)
     - S_2 depends on S_1 is denoted by $S_1 \delta^{-1} S_2$
  3. Output dependence (WAW hazard)
     - S_2 depends on S_1 is denoted by $S_1 \delta^{0} S_2$

Dependences

- Formally:

  There is a data dependence from statement S_1 to statement S_2 (S_2 depends on S_1) if:
  1. Both statements access the same memory location
  2. At least one of them stores onto it, and
  3. There is a feasible run-time execution path from S_1 to S_2

The Big Picture

What are our goals?

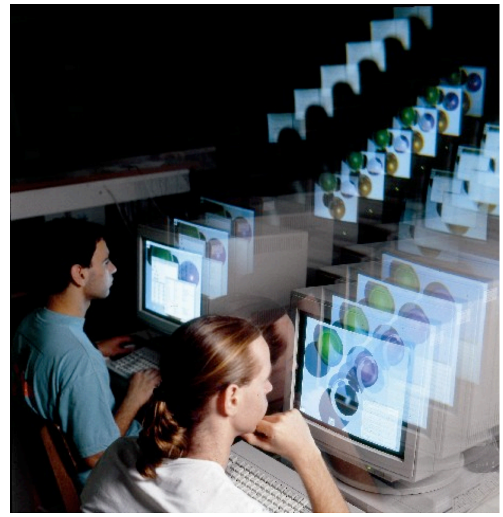
- Simple Goal: Make execution time as short as possible

Which leads to:

- Achieve execution of many (all, in the best case) instructions in parallel
- Find independent instructions

Dependence is Needed in Parallel Programming

http://www.fincher.org/images/2008-04-30-Parallel.jpg



http://www.admin.technion.ac.il/pard/archives/Researchers/ParallelComputing.jpg

Can You Parallelize This?

```
DO I = 1, 100
S1    X(I) = Y(I) + 10
      DO J = 1, 100
S2       B(J) = A(J,N)
         DO K = 1, 100
S3        A(J+1,K)=B(J)+C(J,K)
         ENDDO
S4       Y(I+J) = A(J+1, N)
      ENDDO
ENDDO
```

```
DO I = 1, 100
  DO J = 1, 100
    B(J) = A(J,N)
    A(J+1,1:100)=B(J)+C(J,1:100)
  ENDDO
  Y(I+1:I+100) = A(2:101,N)
ENDDO

X(1:100) = Y(1:100) + 10
```


Dependence in Loops

- Let us look at two different loops:

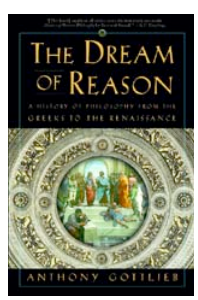
```
    DO I = 2, N-1
S1    A(I+1) = A(I) + B(I)
    ENDDO
```

```
    DO I = 2, N-1
S1    A(I-1) = A(I) + B(I)
    ENDDO
```

- **In both cases, statement S_1 depends on itself**
- ***One can be vectorized (which one?).*** **The other cannot.**
- **We need a formalism to describe and distinguish such dependences**

A Digression: Thesis Writing

- Worry about finding a job or getting a research grant
  - the wrong way to think about it
  - the right way to think about it
- Rhetoric by Aristotle [McCroskey, 1978]
  - attention and good will
  - factual info
  - summary of main points
  - constructive
  - rebuttal
  - conclusion
  - or in Greek (or Latin?)
    - exordium, narratio, divisio, confirmatio, confutatio, conclusio



"it usually starts by precisely defining a subject matter and says which question he is about to answer, then he looks at earlier answers, untangles them, and weighs up objections, he draws distinctions to try to unstitch ambiguouties and confusions, gives his verdict on what problems remain to be solved, argue for an answer of his own, says where its limitations lie and relates to his accounts to other topics, and recap."

***Dream of Reason*, Anthony Gottleb**

What is Philosophy?

- Pythagoras "Lovers of knowledge"
- "Being a systematic spirit without a system"
- To find answer in the basic questions about the world and ourselves in the world.
- Philosophical questions cannot be answered by scientific methods and experimental verification.
- It is what philosophers do. It is something you have to do it to understand it.

A Writing Program

- Write K chapters each on an important subject
- Write clearly and convincingly for each chapter
- Structure of the program
  - loop 1: write chapters, 1 to k
  - loop 2: write sections, intro, body, trans
- How many ways can you write a nested loop?


```
k = 3
k.each do |chap|
    ["intro", "body", "trans"].each do |sec|
        puts "write section "+ sec + " of chapter "+chap.to_s
    end
end

k = 3
for (chap in 1:k)
  for (sec in c("intro", "body", "trans"))
    print(sprintf("write section %d of chapter %s\n", sec, chap))
```

- Which syntax do you like more?
- Can we write in a different order?
  - later chapters may depend on the first
  - intro/trans of next chapter depends on trans of previous
- The problem is first representation
  - representing iterations and their ordering constraints

Representing Iterations

- Iteration Space: The set of all possible iteration vectors for a statement

Example:

```
DO I = 1, 2
   DO J = 1, 2
S1       <some statement>
   ENDDO
ENDDO
```

- The iteration space for S_1 is { (1,1), (1,2), (2,1), (2,2) }

Formal Definition of Loop Dependence

- Theorem 2.1 Loop Dependence:

  There exists a dependence from statements S_1 to statement S_2 in a common nest of loops if and only if there exist two iteration vectors i and j for the nest, such that
  (1) i < j or i = j and there is a path from S_1 to S_2 in the body of the loop,
  (2) statement S_1 accesses memory location **M** on iteration i and statement S_2 accesses location **M** on iteration j, and
  (3) one of these accesses is a write.

- Follows from the definition of dependence

Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program

- But, what is the "meaning" of a program?

For our purposes:

- Two computations are equivalent if, on the same inputs:
  - They produce the same outputs in the same order

Reordering Transformations

- A reordering transformation is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statements

19

Properties of Reordering Transformations

- A reordering transformation does not eliminate dependences
- However, it can change the ordering of the dependence which will lead to incorrect behavior
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

20

Fundamental Theorem of Dependence

- Fundamental Theorem of Dependence:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program
- Proof by contradiction. Theorem 2.2 in the book.

21

Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program
- But, what is the "meaning" of a program?

For our purposes:

- Two computations are equivalent if, on the same inputs:
  - They produce the same outputs in the same order

22

Reordering Transformations

- A reordering transformation is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statements

23

Properties of Reordering Transformations

- A reordering transformation does not eliminate dependences
- However, it can change the ordering of the dependence which will lead to incorrect behavior
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

24

Fundamental Theorem of Dependence

- Fundamental Theorem of Dependence:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program
- Proof by contradiction. Theorem 2.2 in the book.

Distance and Direction Vectors

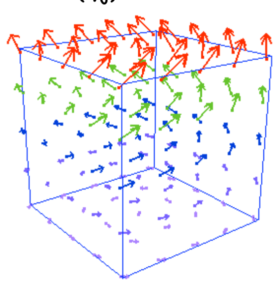
- Consider a dependence in a loop nest of n loops
  - Statement S_1 on iteration i is the source of the dependence
  - Statement S_2 on iteration j is the sink of the dependence
- The distance vector is a vector of length n d(i,j) such that: $d(i,j)_k = j_k - i_k$
- We shall normalize distance vectors for loops in which the index step size is not equal to 1.

Direction Vectors

- Definition 2.10 in the book:

  Suppose that there is a dependence from statement S_1 on iteration **i** of a loop nest of **n** loops and statement S_2 on iteration j, then the **dependence direction vector** is D(i,j) is defined as a vector of length **n** such that

$$D(i,j)_k = \begin{cases} \text{“<” if } d(i,j)_k > 0 \\ \text{“=” if } d(i,j)_k = 0 \\ \text{“>” if } d(i,j)_k < 0 \end{cases}$$



http://www.sciencesoftware.com/ProductImagesA/WebVectorPlot.png

Direction Vectors

Example:

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
S1          A(I+1, J, K-1) = A(I, J, K) + 10
      ENDDO
   ENDDO
ENDDO
```

- S_1 has a true dependence on itself.
- Distance Vector: (1, 0, -1)
- Direction Vector: (<, =, >)

Direction Vectors

- A dependence cannot exist if it has a direction vector whose leftmost non "=" component is not "<" as this would imply that the sink of the dependence occurs before the source.


```
k = 3
k.each do |chap|
   ["intro", "body", "trans"].each do |sec|
      puts "write section "+ sec + " of chapter "+chap.to_s
   end
end
```

```
k = 3
for (chap in 1:k)
  for (sec in c("intro", "body", "trans"))
    print(sprintf("write section %d of chapter %s\n", sec, chap))
```

- What are the distance/direction vectors?
  - later chapters may depend on the first
  - trans of the next depends on trans of the previous
  - intro of the next depends on trans of the previous
  - trans of a chapter depends on its intro

Loop-carried and Loop-independent Dependences

- If in a loop statement S_2 depends on S_1, then there are two possible ways of this dependence occurring:

1. S_1 and S_2 execute on different iterations
   - This is called a loop-carried dependence.
2. S_1 and S_2 execute on the same iteration
   - This is called a loop-independent dependence.

- Loop-independent and loop-carried dependence partition all possible data dependences!

Loop-carried dependence

- Definition 2.11
  - Statement S_2 has a **loop-carried dependence** on statement S_1 if and only if D(i,j) contains a "<" as leftmost non "=" component.
  - Level of a loop-carried dependence is the index of the leftmost non-"=" of D(i,j) for the dependence.

Example:

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

- Dependence and direction vector?
- Dependence level?

Loop-carried dependence

Another example:

```
   DO I = 1, 10
      DO J = 1, 10
         DO K = 1, 10
S1          A(I, J, K+1) = A(I, J, K)
         ENDDO
      ENDDO
   ENDDO
```

- Dependence and direction vector?
- Dependence level?
- A level-k dependence between S_1 and S_2 is denoted by $S_1 \delta_k S_2$

Loop-carried Transformations

- Theorem 2.4 Any reordering transformation that does not alter the relative order of any loops in the nest and preserves the iteration order of the level-**k** loop preserves all level-**k** dependences.
- Proof:
  - D(i, j) has a "<" in the k^{th} position and "=" in positions 1 through k-1
    ⇒ Source and sink of dependence are in the same iteration of loops 1 through k-1
    ⇒ Cannot change the sense of the dependence by a reordering of iterations of those loops

Statement Reordering

Example:

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

can it be transformed to?

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


Loop-independent dependences

- Definition 2.14. Statement S_2 has a **loop-independent dependence** on statement S_1 if and only if D(i,j) contains only "=" components.

Example:

```
   DO I = 1, 10
S1      A(I) = ...
S2      ... = A(I)
   ENDDO
```

Loop-independent dependences

More complicated example:

```
DO I = 1, 9
S1       A(I) = ...
S2      ...    = A(10-I)
ENDDO
```

- No common loop is necessary. For instance:

```
DO I = 1, 10
S1      A(I) = ...
ENDDO
DO I = 1, 10
S2      ...    = A(20-I)
ENDDO
```


Loop-independent dependences

- Theorem 2.5. If there is a loop-independent dependence from S_1 to S_2, any reordering transformation that does not move statement instances between iterations and preserves the relative order of S_1 and S_2 in the loop body preserves that dependence.

- S_2 depends on S_1 with a loop independent dependence is denoted by $S_1 \, \delta_\infty \, S_2$

Review Questions

- What is data dependence?
- What is the fundamental theorem of dependence?
  - Why is it useful?
- How to represent iterations in a loop nest?
- How to represent loop dependences between iterations?