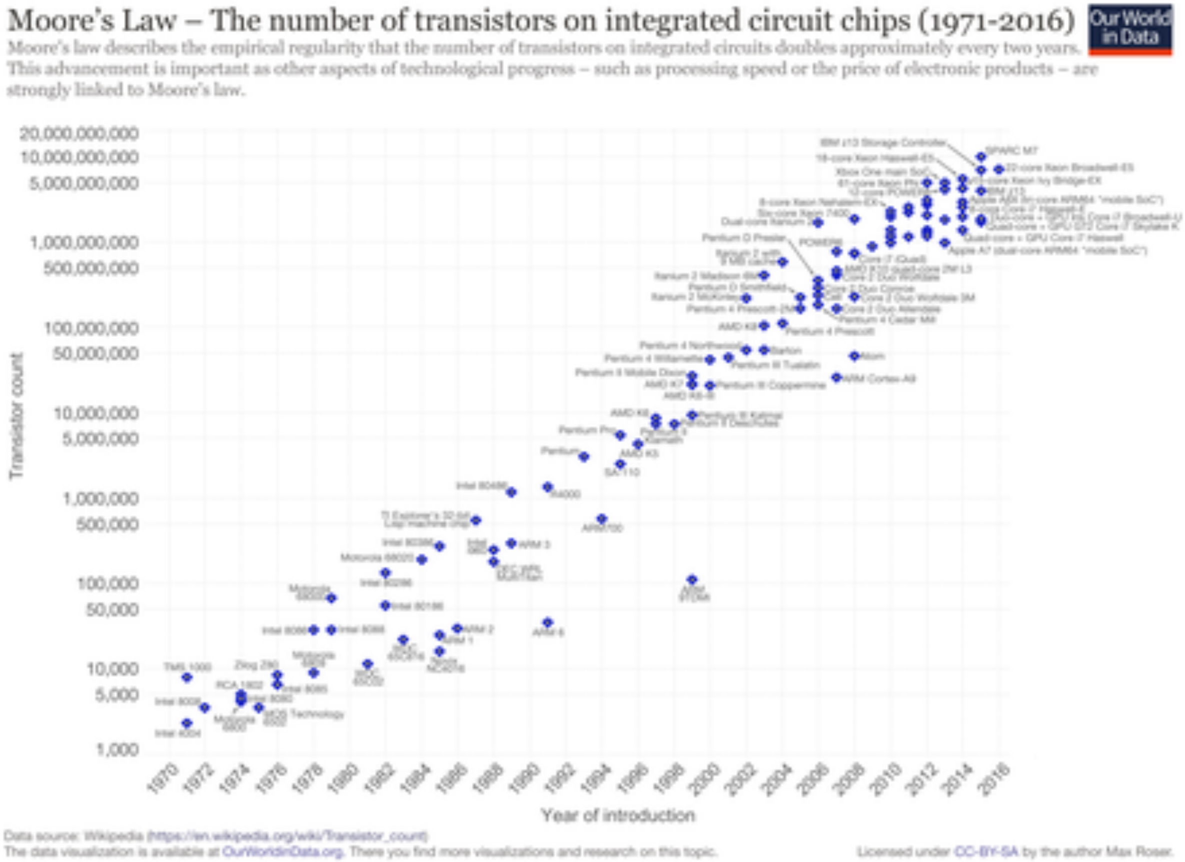
CS 293S Parallelism and Dependence Theory

Yufei Ding

Reference Book:
"Optimizing Compilers for Modern Architecture"
by Allen & Kennedy

Slides adapted from Louis-Noël Pouche, Mary Hall

End of Moore's Law necessitate parallel computing


Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

- End of Moore's law necessitate a means of increasing performance beyond simply producing more complex chips.
  - One such method is to employ cheaper and less complex chips in parallel architectures

Amdahl's law

☐ if *f* is the fraction of the code parallelized, and if the parallelized version runs on a p-processor machine with no communication or parallelization overhead, the speedup is

$$\frac{1}{(1-f)+(f/p)}$$

If f = 50%, than the maximum speedup would be ?

Data locality

- Temporal locality occurs when the same data is used several times within a short time period.
- Spatial locality occurs when different data elements that are located near to each other are used within a short period of time.

- Better locality → less cache misses
- An important form of spatial locality occurs when all the elements that appear on one cache line are used together.

1. Parallelism and data locality are often correlated.
2. Same/Similar set of Techniques for exploring parallelism and maximizing data locality.

Data locality

- Kernels can often be written in many semantically equivalent ways but with widely varying data localities and performances

```
for (j=1; j<N; j++)
    for (i=1; i<N; i++)
        A[i, j] = 0;
```

(a) Zeroing an array column-by-column

```
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        A[i, j] = 0;
```

(b) Zeroing an array row-by-row.

```
b = ceil (N/M)
for (i= b * p; i < min(n, b*(p+1)); i++)
    for (j=1; j<N; j++)
        A[i, j] = 0;
```

(c) Zeroing an array row-by-row in parallel.

A[1,1]	A[2,1]	…	A[1,2]	A[2,2]	…	A[1,3]	…

How to get efficient parallel programs?

- Programmer: writing correct and efficient sequential programs is not easy; writing parallel programs that are correct and efficient is even harder.
  - data locality, data dependence
  - Debugging is hard

- Compiler?
  - Correctness V.S. Efficiency
  - Simple assumption
    - no pointers and pointer arithmetic
    - Affine: Affine loop + affine array access + …

Affine Array Accesses

- Common patterns of data accesses: (i, j, k are loop indexes)
  - A[i], A[j], A[i-1], A[0], A[i+j], A[2*i], A[2*i+1] , A[i,j], A[i-1, j+1]
- Array indexes are affine expressions of surrounding loop indexes
  - Loop indexes: i_n, i_{n-1}, ... , i_1
  - Integer constants: c_n, c_{n-1}, ... , c_0
  - Array index: $c_n i_n + c_{n-1} i_{n-1} + \ldots + c_1 i_1 + c_0$
- Affine expression: linear expression + a constant term (c_0)

Affine loop

- All loop bounds and contained control conditions have to be expressible as a linear affine expression in the containing loop index variables

- Affine array accesses

- No pointers + no possible aliasing (e.g., overlap of two arrays) between statically distinct base addresses.

Loop/Array Parallelism

```
for (i=1; i<N; i++)
   C[i] = A[i]+B[i];
```

- The loop is parallelizable because each iteration accesses a different set of data.
- We can execute the loop on a computer with N processors by giving each processor an unique ID p = 0 , 1 , . . . , M - 1 and having each processor execute the same code:

```
C[p] = A[p]+B[p];
```

Parallelism & Dependence

```
for (i=1; i<N; i++)
   A[i] = A[i-1]+B[i];
```

```
A[1] = A[0]+B[1];
A[2] = A[1]+B[2];
A[3] = A[2]+B[3];
…
```

Focus of the this lecture

- Data Dependence
  - True, Anti-, Output dependence
  - Source and Sink
  - Distance vector, direction vector
  - Relation between Reordering transformation and Direction vector
  - Loop dependence
    - loop-carried dependence
    - Loop-Independent Dependences
  - Dependence graph

Dependence Concepts

Assume statement S_2 depends on statement S_1.

1. True dependences (RAW hazard): read after write.
   Denoted by $S_1 \; d \; S_2$
2. Antidependence (WAR hazard): write after read.
   Denoted by $S_1 \; d^{-1} \; S_2$
3. Output dependence (WAW hazard): write after write.
   Denoted by $S_1 \; d^0 \; S_2$

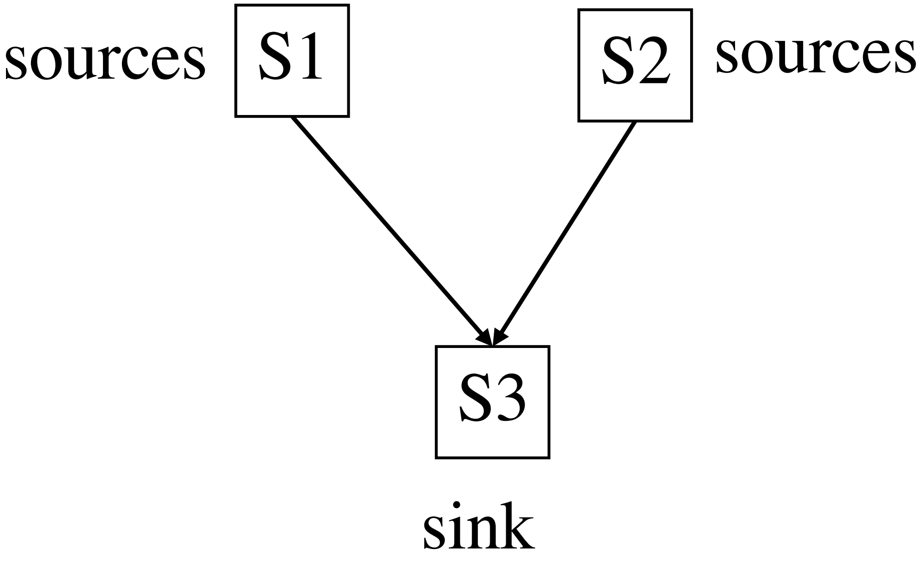
Dependence Concepts

- Source and Sink
  - Source: the statement (instance) executed earlier
  - Sink: the statement (instance) executed later
  - Graphically, a dependence is an edge from source to sink

S_1 PI = 3.14

S_2 R = 5.0

S_3 AREA = PI * R ** 2



Dependence in Loops

☐ Let us look at two different loops:

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

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

- In both cases, statement S_1 depends on itself
- However, there is a significant difference
- We need a formalism to describe and distinguish such dependences

Data Dependence Analysis

Objective: compute the set of statement instances which are dependent

Possible approaches:

- Distance vector: compute an indicator of the distance between two dependent iteration
- Dependence polyhedron: compute list of sets of dependent instances, with a set of dependence polyhedra for each pair of statements

Program Abstraction Level

- Statement

```
For (i = 1; i <=10; i++)
       A[i] = A[i-1] + 1
```

- Instance of statement

```
A[4] = A[3] + 1
```

Iteration Domain

- Iteration Vector
  - A n-level loop nest can be represented as a n-entry vector, each component corresponding to each level loop iterator

```
For ($x_1$=$L_1$; $x_1$<$U_1$; $x_1$++)
    …
    For ($x_2$=$L_2$; $x_2$<$U_2$; $x_2$++)
        …
        For ($x_n$=$L_n$; $x_n$<$U_n$; $x_n$++)
            <some statement $S_1$>
```

$$\vec{x} = \begin{pmatrix} x1 \\ x2 \\ . \\ . \\ . \\ xn \end{pmatrix}$$

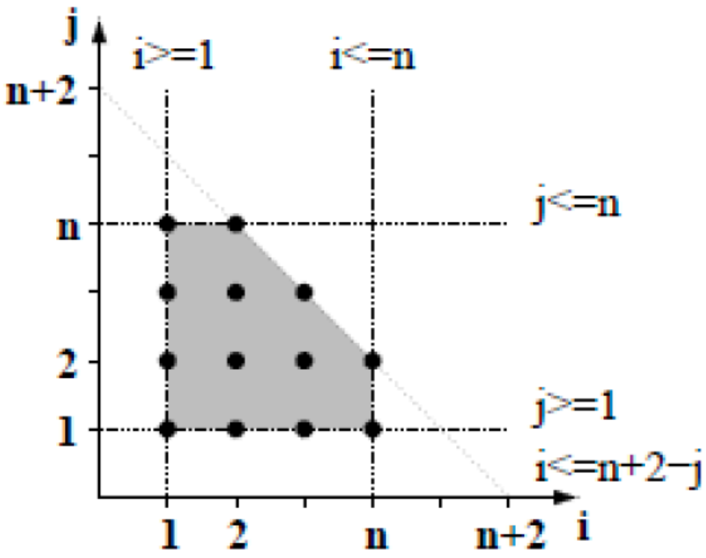
The iteration vector (2, 1, …) denotes the instance of S_1 executed during the 2nd iteration of the X_1 loop and the 1st iteration of the X_2 loop

Iteration Domain

- Dimension of Iteration Domain: Decided by loop nesting levels
- Bounds of Iteration Domain: Decided by loop bounds
  - Using inequalities

```
For (i=1; i<=n; i++)
        For (j=1; j<=n; j++)
                if (i<=n+2-j)
                        b[j]=b[j]+a[i];
```

$$1 \leq i \leq n\,, 1 \leq j \leq n$$
$$i \leq n + 2 - j$$



Modeling Iteration Domains

- ☐ Representing iteration bounds by affine function:

$$1 \le i \le n \;:\; \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \ge 0$$

$$1 \le j \le n \;:\; \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \ge 0$$

$$\mathrm{i} \le n + 2 - j \;:\; \begin{bmatrix} -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + (n + 2) \ge 0$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n + 2 \end{pmatrix} \ge \vec{0}$$

Loop Normalization

- Algorithm:
  - Replace loop boundaries and steps:

    for (i = L, i < U, i = i + S) → for (i = 1, i < (U-L+S)/S, i = i + 1)
  - Replace each reference to original loop variable i with:

    i * S - S + L

Examples: Loop Normalization

```
For (i=4; i<=N; i+=6)
    For (j=0; j<=N; j+=2)
        A[i] = 0
```

```
For (ii=1; ii<=(N+2)/6; ii++)
    For (jj=1; jj<=(N+2)/2; jj++)
        i=ii*6-6+4
        j=jj*2-2
        A[i]=0
```

Distance/Direction Vectors

- The distance vector is a vector d(sink, source) such that:
  - $d_k = sink_k - source_{k.}$
  - i.e., the difference between their iteration vectors
  - sink - source!!
- The direction vector is a vector D(i,j) such that:
  - $D_k =$ "<" if $d(i,j)_k > 0$;
  - $D_k =$ ">" if $d(i,j)_k < 0$;
  - $D_k =$ "=" otherwise.

Example 1:

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

- Dependence distance vector of the <u>true dependence</u>:
  source (write): A(I+1); sink (read): A(I)
- Consider a memory location A(4)
  iteration vector of source: (3)
  iteration vector of sink: (4)
- Distance vector: (4) – (3) = (1)
- Direction vector: (<)

Example 1:

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

- Dependence distance vector of the <u>true dependence</u>:
  source (write): A(I+1); sink (read): A(I)
- More general reasoning:
- Consider a memory location A(x)
  iteration vector of source: (x-1)
  iteration vector of sink: (x)
- Distance vector: (x) - (x-1) = (1)
- Direction vector: (<)

Example 2:

```
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
```

- What is the dependence distance vector of the true dependence?
- What is the dependence distance vector of the anti-dependence?

Example 2:

```
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
```

- ☐ For the true dependence:

  Distance Vector: (1, 0, -1)

  Direction Vector: (<, =, >)

- ☐ For the anti-dependence:

  Distance Vector: (-1, 0, 1)

  Direction Vector: (>, =, <)

sink happens before source: the assumed anti-dependence is invalid!

Example 3:

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

- What is the dependence distance vector of the true dependence?
- What is the dependence distance vector of the anti-dependence?

Example 3:

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

- For the true dependence:
  - Distance Vector: (-1, 0, 1)
  - Direction Vector: (>, =, <)
- For the anti-dependence:
  - Distance Vector: (1, 0, -1)
  - Direction Vector: (<, =, >)

The assumed true dependence is invalid!

Example 2

```
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
```

Example 3

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

- True dependence turns into an anti-dependence.
  “Write then read” turns into “read then write”.
- Reflected in direction vector of the true dependence:
  (<, =, >)   turns into (>, =, <)

Example 4:

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

- What is the dependence distance vector of the true dependence?
- What is the dependence distance vector of the anti-dependence?
- Is this program equivalent with Example 2?

Example 2

```
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
```

Example 4

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

Consider the true dependence

distance vectors	*direction vectors*
(1, 0, -1)	source (<, =, >) sink write read
(0, 1, -1)	source (=, <, >) sink write read

So, it is still a true dependence.

- True dependence stays as true dependence.
  “Write then read” stays as “Write then read”.
- Reflected in direction vector of the true dependence:
  (<, =, >) turns into (=, <, >)

Reordering Transformations

- Definition:
  - merely changes the order of execution of the code
  - no adding or deleting

- A reordering transformation does not eliminate dependences
- However, it can change the execution order of original sink and source, causing incorrect behavior

- “Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.”

---- *Fundamental Theorem of Dependence*

Theorem of loop reordering

- Direction Vector Transformation
  - Let T be a reordering transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop.
  - Then the transformation is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the original nest has a leftmost non- “=” component that is “>”.

- Follows from Fundamental Theorem of Dependence:
  - All dependences exist
  - None of the dependences have been reversed

Procedure to Check Validity of a Loop Reordering

1. List the ***direction vectors of all types of data dependences*** in the original program
2. According to the new order of loops, ***exchange the elements in the direction vectors*** to derive the new direction vectors.
3. If all the direction vectors have **a "<" as the first non-"=" sign**, the transformation is valid.

   A all-"=" vector will stay as all-"=" vector; it won't affect the correctness of loop reordering.

Example

```
DO H = 1, 10
    DO I = 1, 10
        Do J = 1, 10
            Do K = 1, 10
S                   A(H, I+1, J-2, K+3) = A(H, I, J, K) + B
            ENDDO
        ENDDO
    ENDDO
ENDDO
```

?

```
DO H = 1, 10
    DO J = 1, 10
        Do I = 1, 10
            Do K = 1, 10
S                   A(H, I+1, J-2, K+3) = A(H, I, J, K) + B
            ENDDO
        ENDDO
    ENDDO
ENDDO
```