CS5375 Computer Systems Organization and Architecture

Lecture 19

Instructor: Yong Chen, Ph.D.
Department of Computer Science
Texas Tech University
Yong.Chen@ttu.edu, 806-834-0284

Announcements

- Programming project #2 posted and due by 12/6, Tue., 11:59 p.m.

- Recording by Mr. Mert Side for two ways of running a GPU job on TTU systems for programming project #2
  - https://www.youtube.com/watch?v=5wIF0aVAMD0
  - Approach #1: through the interactive session on the gpu-build node (not preferred, will not be available if too many users are using the interactive session simultaneously)
  - Approach #2: though the Slurm batch job scheduler (preferred, can handle a large number of simultaneous jobs)

Outline

- Parallelization and Loop-Level Parallelism

When Can 2 Statements Execute in Parallel?

- On one processor:

    statement 1;

    statement 2;

- On two processors:

processor1:	processor2:
statement1;	statement2;

When Can 2 Statements Execute in Parallel? (cont.)

- Possibility #1

Processor1: Processor2:

statement1;

statement2;

time

- Possibility #2

Processor1: Processor2:

statement2:

statement1;

time

When Can 2 Statements Execute in Parallel? (cont.)

- Their order of execution must not matter!

- In other words,

statement1; statement2;

must be equivalent to

statement2; statement1;

Example 1

a = 1;

b = 2;

Yes!

Example 2

Assuming a has value of 0 initially

```
a = 1;
b = a;
```

No!

Example 3

Assuming a has value of 0 initially

```
b = a;
a = 1;
```

No!

Example 4

Assuming a has value of 0 initially

a = 1;

a = 2;

Write after write

No!

When Can 2 Statements Execute in Parallel? (cont.)

- S1 and S2 can execute in parallel

  iff?

There are no dependences between S1 and S2

- Data dependences
  - True dependences
  - Anti-dependences
  - Output dependences
  - Loop-carried dependences
- Control dependences

Data Dependence

- Assuming statement S1 and S2, S2 depends on S1 if:
  $[O(S1) \cap I(S2)] \cup [I(S1) \cap O(S2)] \cup [O(S1) \cap O(S2)] \neq \emptyset$ non of the three are not empty

  O means output set of memory location used by mem

  Where:

  I(Si) is the set of memory locations read by Si and
  O(Sj) is the set of memory locations written by Sj
  and there is a feasible run-time execution path from S1 to S2

- Three cases

True Dependence

Statements S1, S2

S2 has a true dependence on S1

if $O(S1) \cap I(S2) \neq \emptyset$

what varibales s1 writes, same variables read by s2

Here s1={a} which means output of statement 1
s2={a} which means input for statement 2
{a} intersection {a} != empty

S1 has a write and is followed by a read of the same location in S2 (read after write)

```
a = 1;
b = a;
```

Anti-Dependence

Statements S1, S2.

S2 has an anti-dependence on S1

if $I(S1) \cap O(S2) \neq \emptyset$, mirror relationship of true dependence

{a} intersection {a} = {a}

S1 has a read and is followed by a write to the same location in S2 (write after read)

Anti-dependence can be removed with renaming technique

```
b = a;
a = 1;
```

This is write after read

Output Dependence

Statements S1, S2.

S2 has an output dependence on S1

if $O(S1) \cap O(S2) \neq \emptyset$

{a} interection {a} != empty

S1 has a write and is followed by a write to the same location in S2 (write after write)

Output dependence can be removed with renaming technique

```
a = 1;
a = 2;
```

This is write after write

Example 5

- Most parallelism occurs in loops, that is what GPU architecture, vector architecture designed for

```
for(i=0; i<100; i++)
    a[i] = i;
```

Yes!

Example 6

```
for(i=0; i<100; i++) {
    a[i] = i;
    b[i] = 2*i;
}
```

Yes!

Example 7

```
for(i=0;i<100;i++) a[i] = i;
for(i=0;i<100;i++) b[i] = 2*i;
```

Yes!

Example 8

```
for(i=0; i<100; i++)
    a[i] = a[i] + 100;
```

Yes!

Example 9

```
for( i=1; i<100; i++ )
  a[i] = f(a[i-1]);
```

```
a[1] = function(a[1])
a[2] = function(a[2])
```

No!

Loop-Carried Dependence

- A loop-carried dependence is a dependence between instructions from different iterations of a loop

- Otherwise, we call it a loop-independent code

- Loop-carried dependences prevent loop iteration parallelization

- Loop-carried dependences can sometimes be removed with loop interchange

Loop-Level Parallelism

- Example 10 (page 339, Chapter 4.5):

```
for (i=999; i>=0; i=i-1)
 x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism

- Example 11 (page 339, Chapter 4.5):

```
for (i=0; i<100; i=i+1) {
 A[i+1] = A[i] + C[i]; /* S1 */
 B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

True Depedecy = READ AFTER WRITE
Anti Dependecy = WRITE AFTER READ

we have dependecy here
read after write

- S1 and S2 use values computed by S1 in the previous iteration (loop-carried)

s1 uses the values which are generated in the previous iteration

- S2 uses value computed by S1 in the same iteration (not loop-carried)

Example 12

```
for(i=0; i<100; i++ )
  for(j=1; j<100; j++ )
    a[i][j] = f(a[i][j-1]);
```

No!

Example 13

```
for( j=1; j<100; j++ )
  for( i=0; i<100; i++ )
    a[i][j] = f(a[i][j-1]);
```

Yes for
i loop!

Example 14 (page 340)

```
for (i=0; i<100; i=i+1) {
  A[i] = A[i] + B[i]; /* S1 */
  B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration (loop-carried dependence), but loop is parallel with a transformation
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
  B[i+1] = C[i] + D[i];
  A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

i=0 s1: A[0] = A[o]+B[o]

Finding and Removing Dependencies

- A comprehensive example
- Example 15: (page 343)

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- True dependences from S1 to S3 and from S1 to S4 because of Y[i] read after write
  - Not loop carried, do not prevent the loop from being considered parallel
  - These dependences will force S3 and S4 to wait for S1 to complete
- Anti-dependence from S1 to S2, based on X[i] write after read
- Anti-dependence from S3 to S4 for Y[i] write after read
- Output dependence from S1 to S4, based on Y[i] read after write and write after write

Finding and Removing Dependencies

- After renaming

```
for (i=0; i<100; i=i+1 {
  T[i] = X[i] / c;   /* Y renamed to T to remove output dependence */
  X1[i] = X[i] + c;/* X renamed to X1 to remove anti-dependence */
  Z[i] = T[i] + c;  /* Y renamed to T to remove anti-dependence */
  Y[i] = c - T[i];
}
```

Control Dependence

- An instruction is control dependent on a preceding instruction if the outcome of latter determines whether former should be executed or not

- S2 is control dependent on instruction S1. However, S3 is not control dependent upon S1
  - S1. if (a == b)
  - S2. a = a + b
  - S3. b = a + b

Control Dependence

- There is control dependence b.t. statements S1 and S2 if
  - S1 could be possibly executed before S2
  - The outcome of S1 will determine whether S2 will be executed

- A typical example is that there is control dependence between if statement's condition part and the statements in the corresponding true/false bodies

Parallelization

- Parallelizing compilers analyze program dependences to decide parallelization
- In parallelization by hand, user does the same analysis

- Compiler more convenient and more correct
- User more powerful, can analyze more patterns

To Remember

- Statement order must not matter
- Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious

Readings

- Chapter 4, 4.5