CS4379: Parallel and Concurrent Programming
CS5379: Parallel Processing

Lecture 5

Dr. Yong Chen

Associate Professor

Computer Science Department

Texas Tech University

Course Info

- **Lecture Time**: TR, 12:30-1:50
- **Lecture Location**: ECE 217
- **Sessions**: CS4379-001, CS4379-002, CS5379-001, CS5379-D01
- **Instructor:** Yong Chen, Ph.D., Associate Professor
- **Email:** yong.chen@ttu.edu
- **Phone:** 806-834-0284
- **Office**: Engineering Center 315
- **Office Hours**: 2-4 p.m. on Wed., or by appointment
- **TA:** Mr. Ghazanfar Ali, Ghazanfar.Ali@ttu.edu
- **TA Office hours:** Tue. and Fri., 2-3 p.m., or by appointment
- **TA Office:** EC 201 A
- **More info:**
  - http://www.myweb.ttu.edu/yonchen
  - http://discl.cs.ttu.edu; http://cac.ttu.edu/; http://nsfcac.org

Announcements/Reminders

- Assignment #1 posted and due in one week
  - Due on 2/6, Thur., 12:30 p.m.
  - Please submit a soft copy via Blackboard (preferred) or a hard copy in class

Outline

- Questions?

- Parallelism
  - Different flavors
  - Data dependence
  - Control dependence
  - Parallelization

Parallelism

- Ability to execute different parts of a program concurrently on different processors

- Goals:
  - Solves a problem concurrently
  - Reduces the time to solution (speedup)
  - Solves larger problems

Flavors of Parallelism

- Two primary types of parallelism: data parallelism and task parallelism
- Data parallelism:
  - Identical operations operate on data items concurrently to solve a problem
- Task parallelism:
  - Independent tasks (non-identical operations) operate on data items concurrently to solve a problem

Data and Task Parallelization

- Data parallel:

```
for (i=0;i<1000;i++)
   a[i]=b[i]+c[i];
```

Suitable for SIMD architecture

- Task parallel:

```
for (i=0;i<1000;i++)  /*block 1 */
   b[i+1]=b[i]+c[i]
…
for (j=0;j<5;j++) /*block 2*/
    a[j+1]=a[j]+d[j];
```

Suitable for MIMD architecture

Coarse and Fine Grain Parallelism

- Grain size categorizes amount of compute work done in relation to communication, i.e., the ratio of computation to the amount of communication

- Fine grain
  - Small computation in terms of code size and execution time
  - Frequent communications
  - Better parallelism and load balance, but greater overheads of synchronization and communication

- Coarse grain: opposite
  - Larger computations, infrequent communications, smaller overheads, less parallelism, more likely load imbalance

- Need a balance to attain the best performance

Dependence & Parallelization

- Consider the following loop of a C program

```
for (i=0;i<1000;i++)
    a[i]=b[i]+c[i]
```

- If one unfolds the loops, the statements would be executed as follows:

```
a[0]=b[0]+c[0];
a[1]=b[1]+c[1];
.....
a[999]=b[999]+c[999];
```

- Can each iteration be executed in parallel?

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?

- Possibility 1

  Processor1: Processor2:

  statement1;

  statement2;

  time

- Possibility 2

  Processor1: Processor2:

  statement2:

  statement1;

  time

When can 2 statements execute in parallel?

- 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

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

No!

Example 3

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

No!

Example 4

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

No!

Data Dependence

- Assuming statement S1 and S2, S2 depends on S1 if:
  $[O(S1) \cap I(S2)] \cup [I(S1) \cap O(S2)] \cup [O(S1) \cap O(S2)] \neq \emptyset$

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

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

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

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

Output Dependence

Statements S1, S2.

S2 has an output dependence on S1
if $O(S1) \cap O(S2) \neq \emptyset$

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

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

Removing Dependences

- Some dependences can be removed
  - Name dependences: when two instructions use the same register or memory location, but no flow of data between the instructions
  - Anti-dependences
  - Output dependences

Removing Anti-dependence

- An anti-dependence is an example of a name dependence
  - That is, renaming variables could remove the dependence

```
1. B = 3
2. A = B + 1
3. B = 7
```

→

```
1. B = 3
N. B2 = B
2. A = B2 + 1
3. B = 7
```

- A new variable, B2, has been declared as a copy of B in a new instruction, instruction N
- The anti-dependence between 2 and 3 has been removed, meaning that these instructions may now be executed in parallel
- However, the modification has introduced a new dependence: instruction 2 is now truly dependent on instruction N, which is truly dependent upon instruction 1.

Removing Output Dependence

- As with anti-dependencies, output dependencies are also name dependencies
  - That is, they may be removed through renaming of variables, as in the below modification of the above example:

```
1 A = 2 * X
2 B = A / 3
3 A = 9 * Y
```

→

```
1 A2 = 2 * X
2 B = A2 /3
3 A = 9 * Y
```

Example 5

- Most parallelism occurs in loops

```
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]);
```

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 some times be removed with loop interchange

Example 10

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

No!

Example 11

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

Yes for *i* loop!

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

- Intuitively, 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

When can 2 statements execute in parallel?

- S1 and S2 can execute in parallel

  iff?

there are no dependences between S1 and S2

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

Parallelization

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

- Compiler more convenient and more accurate
- 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.

Questions?

Questions/Suggestions/Comments are always welcome!

Write me: yong.chen@ttu.edu
Call me: 806-834-0284
See me: ENGCTR 315

If you write me an email for this class, please start the email subject with [CS4379] or [CS5379].