CS61A Lecture 29
Parallel Computing

Jom Magrotker
UC Berkeley EECS
August 7, 2012

TODAY

- Parallel Computing
  - Problems that can occur
  - Ways of avoiding problems

SO FAR

- functions
- data structures
- objects
- abstraction
- interpretation
- evaluation

One Program
One Computer

YESTERDAY & TODAY

- Multiple Programs!
  - On Multiple Computers
    (Networked and Distributed Computing)
  - On One Computer
    (Concurrency and Parallelism)

YESTERDAY: DISTRIBUTED COMPUTING

- Programs on different computers working together towards a goal.
  - Information Sharing & Communication
    Ex. Skype, The World Wide Web, Cell Networks
  - Large Scale Computing
    Ex. “Cloud Computing” and Map-Reduce
REVIEW: DISTRIBUTED COMPUTING

- Architecture
  - Client-Server
  - Peer-to-Peer
- Message Passing
- Design Principles
  - Modularity
  - Interfaces

TODAY: PARALLEL COMPUTATION

Simple Idea: Have more than one piece of code running at the same time.

Question is: why bother?

PARALLEL COMPUTATION: WHY?

Primarily: Speed.

The majority of computers have multiple processors, meaning that we can actually have two (or more) pieces of code running at the same time!

THE LOAD-COMPUTE-STOE MODEL

For today, we will be using the following model for computation.

\[ x = x \times 2 \]

1. Load the variable(s)
2. Compute the right hand side
3. Store the result into the variable.

THE LOAD-COMPUTE-STOE MODEL

1. Load the variable(s)
2. Compute the right hand side
3. Store the result into the variable.

\[ x = x \times 2 \]

1. Load up \( x: x \rightarrow 5 \)
2. Compute \( x \times 2: 10 \)
3. Store the new value of: \( x \leftarrow 10 \)

ANNOUNCEMENTS

- Project 4 due Today.
  - Partnered project, in two parts.
  - Twelve questions, so please start early!
  - Two extra credit questions.
- Homework 14 due Today.
  - Now includes contest voting.
  - Assignment is short.
- Tomorrow at the end of lecture there is a course survey, please attend!
ANNOUNCEMENTS: FINAL

- Final is Thursday, August 9.
  - Where? 1 Pimentel.
  - When? 6PM to 9PM.
  - How much? All of the material in the course, from June 18 to August 8, will be tested.
- Closed book and closed electronic devices.
- One 8.5" x 11" 'cheat sheet' allowed.
- No group portion.
- We have emailed you if you have conflicts and have told us. If you haven’t told us yet, please let us know by yesterday.
- We’ll email you the room later today.
- Final review session 2 Tonight, from 8pm to 10pm in the HP Auditorium (306 Soda).

Parallel Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = x \times 2)</td>
<td>(y = y + 1)</td>
<td>(x: 20) (y: 4)</td>
</tr>
</tbody>
</table>

Idea is that we perform the steps of each thread together, interleaving them in any way we want.

Thread 1: L1, C1, S1
Thread 2: L2, C2, S2

SO WHAT?

Okay, what’s the point?

IDEALLY: It shouldn’t matter what different “shuffling” of the steps we do, the end result should be the same.

BUT, what if two threads are using the same data?

Parallel Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = x + 2)</td>
<td>(x = x + 1)</td>
<td>(x: 21) or (x: 22)</td>
</tr>
</tbody>
</table>

L1: Load X
C1: Compute \(X \times 2\)
S1: Store \((X \times 2) \rightarrow X\)

Thread 1: L1, C1, S1
Thread 2: L2, C2, S2

Parallel Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = x \times 2)</td>
<td>(x = x + 1)</td>
<td>(x: 20)</td>
</tr>
</tbody>
</table>

L1: Load X
C1: Compute \(X \times 2\)
S1: Store \((X \times 2) \rightarrow X\)

Thread 1: L1, C1, S1
Thread 2: L2, C2, S2
OH NOES!!1!one

We got a wrong answer!

This is one of the dangers of using parallelism in a program!

What happened here is that we ran into the problem of non-atomic (multi-step) operations. A thread could be interrupted by another and result in incorrect behavior!

So, smarty pants, how do we fix it?

PRACTICE: PARALLELISM

Suppose we initialize the value $z$ to 3. Given the two threads, which are run at the same time, what are all the possible values of $z$ after they run?

$z = z \times 3$

$y = z + 0$

$z = z + y$

99, 6, 198, 102

LOCKS

We need a way to make sure that only one person messes with a piece of data at a time!

from Threading import Lock

x_lock = Lock()

Try to get exclusive rights to the lock. If succeeds, keep working. Otherwise, wait for lock to be released.

x_lock.acquire()

x = x * 2

x_lock.release()

Give up your exclusive rights so someone else can take their turn.

x_lock.acquire()

x = x + 1

x_lock.release()

SOUNDS GREAT!

Now only one thread will manipulate the value $x$ at a time if we always wrap code touching $x$ in a lock acquire and release.

So our code works as intended! BUT

"With great power comes great responsibility!"
**Misusing Our Power**

```python
from threading import Lock

x_lock1 = Lock()
x_lock2 = Lock()

# THREAD 1
x_lock1.acquire()
x = x * 2
x_lock1.release()

# THREAD 2
x_lock2.acquire()
x = x + 2
x_lock2.release()
```

Won't work! They aren't being locked out using the same lock, we just went back to square 1.

---

**Misusing Our Power**

```python
from threading import Lock

x_lock1 = Lock()
x_lock2 = Lock()

# THREAD 1
x_lock1.acquire()
x = x * 2
x_lock1.release()

# THREAD 2
x_lock2.acquire()
x = x + 1
x_lock2.release()
```

If `LONG_COMPUTATION` doesn't need `x`, we just caused thread 2 to have to wait a LONG time for a bunch of work that could have happened in parallel. This is inefficient!

---

**Misusing Our Power**

```python
from threading import Lock

x_lock = Lock()

# THREAD 1
x_lock.acquire()
sleep(num)
x_lock.acquire()
start_turn(num)
x = x + 1
x_lock.release()

# THREAD 2
y_lock.acquire()
x_lock.acquire()
y, x = 2 * y, 22
y_lock.release()
x_lock.release()
```

If we didn't actually want thread 2 to be less likely to get access to `x` first each time, then this is an unfair solution that favors the first (lowered numbered) threads.

---

**Misusing Our Power**

```python
from threading import Lock

x_lock = Lock()
y_lock = Lock()

# THREAD 1
x_lock.acquire()
y_lock.acquire()
x, y = 2 * x, 22
x_lock.release()
y_lock.release()

# THREAD 2
x_lock.acquire()
y_lock.acquire()
y, x = 2 * x, 22
x_lock.release()
y_lock.release()
```

What if thread 1 acquires the `x_lock` and thread 2 acquires the `y_lock`? Now nobody can do work! This is called **deadlock**.

---

**The 4 Types of Problems**

In general, you can classify the types of problems you see from parallel code into 4 groups:

- Incorrect Results
- Inefficiency
- Unfairness
- Deadlock

---

**How Do We Avoid Issues?**

Honestly, there isn’t a one-size-fits-all solution.

You have to be careful and think hard about what you’re doing with your code.

In later courses (CS162), you learn common conventions that help avoid these issues, but there’s still the possibility that problems will occur!
ANOTHER TOOL: SEMAPHORES

Locks are just a really basic tool available for managing parallel code.

There’s a LARGE variety of tools out there that you might encounter.

For now, we’re going to quickly look at one more classic:

– Semaphores

What if I want to allow only up to a certain number of threads manipulate the same piece of data at the same time?

```
fruit_bowl = ['banana', 'banana', 'banana']
```

```
def eat_thread():
    fruit_bowl.pop()
    print("ate a banana!")
def buy_thread():
    fruit_bowl.append("banana")
    print("bought a banana")
```

A Semaphore is a classic tool for this situation!

```
from threading import Semaphore
fruit_bowl = ['banana', 'banana', 'banana']
fruit_sem = Semaphore(len(fruit_bowl)) #3

def eat_thread():
    fruit_sem.acquire()
    fruit_bowl.pop()
    print("ate a banana!")

def buy_thread():
    fruit_bowl.append("banana")
    print("bought a banana")
    fruit_sem.release()
```

Increments the count
Decrements the counter. If the counter is 0, wait until someone increments it before subtracting 1 and moving on.

PRACTICE: WHAT COULD GO WRONG?

What, if anything, is wrong with the code below?

```
from threading import Lock
x_lock = Lock()
y_lock = Lock()

def thread1():
    global x, y
    the_lock.acquire()
    x = fib(5000)
    the_lock.release()

def thread2():
    global x, y
    y_lock.acquire()
    x, y = y, x
    print(x + y)
    y_lock.release()
```

CONCLUSION

• Parallelistm in code is important for making efficient code!
• Problems arise when we start manipulating data shared by multiple threads.
• We can use locks or semaphores to avoid issues of incorrect results.
• There are 4 main problems that can occur when writing parallel code.
• Preview: A Powerful Pattern for Distributed Computing, MapReduce
EXTRAS: SERIALIZERS

Honestly, this is just a cool way to use locks with functions.

```python
x_serializer = make_serializer()
x = 5
@x_serializer
def thread1():
    global x
    x = x * 20
@x_serializer
def thread2():
    global x
    x = x + 500
```

```python
def make_serializer():
    serializer_lock = Lock()
    def serialize(fn):
        def serialized(*args):
            serializer_lock.acquire()
            result = fn(*args)
            serializer_lock.release()
            return result
        return serialized
    return serialize
```