Introduction

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Moore’s Law

Transistor count still rising

Clock speed flattening sharply
Still on some of your desktops: The Uniprocessor
In the Enterprise: The Shared Memory Multiprocessor (SMP)
Your New Desktop: The Multicore Processor (CMP)

All on the same chip

Sun T2000 Niagara
Multicores Are Here

- “Intel's Intel ups ante with 4-core chip. New microprocessor, due this year, will be faster, use less electricity...” [San Fran Chronicle]
- “AMD will launch a dual-core version of its Opteron server processor at an event in New York on April 21.” [PC World]
- “Sun’s Niagara...will have eight cores, each core capable of running 4 threads in parallel, for 32 concurrently running threads. ....” [The Inquirer]
Why do we care?

• Time no longer cures software bloat
  – The “free ride” is over
• When you double your program’s path length
  – You can’t just wait 6 months
  – Your software must somehow exploit twice as much concurrency
Traditional Scaling Process

Speedup

User code

Traditional Uniprocessor

Time: Moore’s law
Multicore Scaling Process

Unfortunately, not so simple...
Real-World Scaling Process

Parallelization and Synchronization require great care...
Multicore Programming: Course Overview

• Fundamentals
  – Models, algorithms, impossibility
• Real-World programming
  – Architectures
  – Techniques
Multicore Programming: Course Overview

• Fundamentals
  - Models, algorithms
• Real-World programming
  - Architectures
  - Techniques

We don’t necessarily want to make you experts...
Sequential Computation

memory

object

thread

object
Asynchrony

- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)
Model Summary

• Multiple *threads*
  – Sometimes called *processes*
• Single shared *memory*
• *Objects* live in memory
• Unpredictable asynchronous delays
Road Map

- We are going to focus on principles first, then practice
  - Start with idealized models
  - Look at simplistic problems
  - Emphasize correctness over pragmatism
  - “Correctness may be theoretical, but incorrectness has practical impact”
Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads, processes
- Sometimes OK to confuse them, sometimes not.
Parallel Primality Testing

• Challenge
  – Print primes from 1 to $10^{10}$

• Given
  – Ten-processor multiprocessor
  – One thread per processor

• Goal
  – Get ten-fold speedup (or close)
Load Balancing

- Split the work evenly
- Each thread tests range of $10^9$
void primePrint {
    int i = ThreadID.get(); // IDs in \{0..9\}
    for (j = i*10^9+1, j<(i+1)*10^9; j++) {
        if (isPrime(j))
            print(j);
    }
}
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  – Uneven
  – Hard to predict
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  – Uneven
  – Hard to predict
• Need dynamic load balancing
Shared Counter

each thread takes a number
Procedure for Thread \textit{i}

```java
int counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^{10}) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```
Procedure for Thread $i$

```java
Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^10) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```

Shared counter object
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10+1, j<(i+1)*10; j++) {
        if (isPrime(j))
            print(j);
    }
}
Procedure for Thread $i$

Counter counter = new Counter(1);  

void primePrint {
    long j = 0;
    while (j < $10^{10}$) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
Procedure for Thread \( i \)

Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^{10}) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
Counter Implementation

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```
Counter Implementation

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```

OK for single thread, not for concurrent threads
What It Means

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```
What It Means

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp = value;
        value = value + 1;
        return temp;
    }
}
```
Not so good...

Value...

1

2

3

2

Value...

read 1
write 2
read 2
write 3
read 1
write 2

time
Is this problem inherent?

If we could only glue reads and writes...
Challenge

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
```
public class Counter {
    private long value;

    public long getAndIncrement() {
        int temp = value;
        value = temp + 1;
        return temp;
    }
}

Make these steps atomic (indivisible)
Hardware Solution

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp = value;
        value = temp + 1;
        return temp;
    }
}
```

ReadModifyWrite() instruction
An Aside: Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
An Aside: Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            long temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Mutual Exclusion
Mutual Exclusion or “Alice & Bob share a pond”
Alice has a pet
Bob has a pet
The Problem

The pets don’t get along
Formalizing the Problem

• Two types of formal properties in asynchronous computation:
  • Safety Properties
    – Nothing bad happens ever
  • Liveness Properties
    – Something good happens eventually
Formalizing our Problem

• Mutual Exclusion
  – Both pets never in pond simultaneously
  – This is a safety property

• No Deadlock
  – if only one wants in, it gets in
  – if both want in, one gets in.
  – This is a liveness property
Simple Protocol

• Idea
  – Just look at the pond
• Gotcha
  – Trees obscure the view
Interpretation

• Threads can’t “see” what other threads are doing
• Explicit communication required for coordination
Cell Phone Protocol

• Idea
  – Bob calls Alice (or vice-versa)

• Gotcha
  – Bob takes shower
  – Alice recharges battery
  – Bob out shopping for pet food …
Interpretation

• Message-passing doesn’t work
• Recipient might not be
  – Listening
  – There at all
• Communication must be
  – Persistent (like writing)
  – Not transient (like speaking)
Can Protocol
Bob conveys a bit
Bob conveys a bit
Can Protocol

• Idea
  - Cans on Alice’s windowsill
  - Strings lead to Bob’s house
  - Bob pulls strings, knocks over cans

• Gotcha
  - Cans cannot be reused
  - Bob runs out of cans
Interpretation

• Cannot solve mutual exclusion with interrupts
  - Sender sets fixed bit in receiver’s space
  - Receiver resets bit when ready
  - Requires unbounded number of interrupt bits
Flag Protocol

Art of Multiprocessor Programming
Alice’s Protocol (sort of)
Bob’s Protocol (sort of)
Alice’s Protocol

• Raise flag
• Wait until Bob’s flag is down
• Unleash pet
• Lower flag when pet returns
Bob’s Protocol

• Raise flag
• Wait until Alice’s flag is down
• Unleash pet
• Lower flag when pet returns
Bob’s Protocol (2nd try)

- Raise flag
- While Alice’s flag is up
  - Lower flag
  - Wait for Alice’s flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns
Bob’s Protocol

• Raise flag
• While Alice’s flag is up
  – Lower flag
  – Wait for Alice’s flag to go down
  – Raise flag
• Unleash pet
• Lower flag when pet returns
The Flag Principle

• Raise the flag
• Look at other’s flag
• Flag Principle:
  – If each raises and looks, then
  – Last to look must see both flags up
Proof of Mutual Exclusion

• Assume both pets in pond
  – Derive a contradiction
  – By reasoning **backwards**
• Consider the last time Alice and Bob each looked before letting the pets in
• Without loss of generality assume Alice was the last to look…
Proof

Bob last raised flag

Alice last raised her flag

Bob’s last look

Alice’s last look

Alice must have seen Bob’s Flag. A Contradiction

QED
Proof of No Deadlock

• If only one pet wants in, it gets in.
Proof of No Deadlock

• If only one pet wants in, it gets in.
• Deadlock requires both continually trying to get in.
Proof of No Deadlock

• If only one pet wants in, it gets in.
• Deadlock requires both continually trying to get in.
• If Bob sees Alice’s flag, he gives her priority (a gentleman...)

QED
Remarks

• Protocol is *unfair*
  - Bob’s pet might never get in

• Protocol uses *waiting*
  - If Bob is eaten by his pet, Alice’s pet might never get in
Moral of Story

• Mutual Exclusion cannot be solved by
  – transient communication (cell phones)
  – interrupts (cans)

• It can be solved by
  – one-bit shared variables
  – that can be read or written
The Arbiter Problem (an aside)

Pick a point

Pick a point
The Fable Continues

• Alice and Bob fall in love & marry
The Fable Continues

• Alice and Bob fall in love & marry
• Then they fall out of love & divorce
  - She gets the pets
  - He has to feed them
The Fable Continues

• Alice and Bob fall in love & marry
• Then they fall out of love & divorce
  – She gets the pets
  – He has to feed them
• Leading to a new coordination problem: Producer-Consumer
Bob Puts Food in the Pond
Alice releases her pets to Feed
Producer/Consumer

• Alice and Bob can’t meet
  – Each has restraining order on other
  – So he puts food in the pond
  – And later, she releases the pets

• Avoid
  – Releasing pets when there’s no food
  – Putting out food if uneaten food remains
Producer/Consumer

• Need a mechanism so that
  – Bob lets Alice know when food has been put out
  – Alice lets Bob know when to put out more food
Surprise Solution
Bob puts food in Pond
Bob knocks over Can
Alice Releases Pets

cola

yum…
yum…
Alice Resets Can when Pets are Fed
Pseudocode

Alice’s code

```java
while (true) {
    while (can.isUp())
    {
        pet.release();
        pet.recapture();
        can.reset();
    }
}
```
Pseudocode

Alice's code

```cpp
while (true) {
    while (can.isUp()){};
    pet.release();
    pet.recapture();
    can.reset();
}
```

Bob's code

```cpp
while (true) {
    while (can.isDown()){};
    pond.stockWithFood();
    can.knockOver();
}
```
Correctness

• Mutual Exclusion
  – Pets and Bob never together in pond
Correctness

• Mutual Exclusion
  - Pets and Bob never together in pond

• No Starvation
  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.
Correctness

• **Mutual Exclusion**

  - Pets and Bob never together in pond

• **No Starvation**

  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.

• **Producer/Consumer**

  The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.
Could Also Solve Using Flags

Art of Multiprocessor Programming
Waiting

• Both solutions use waiting
  – \texttt{while(mumble)}{}

• Waiting is \textit{problematic}
  – If one participant is delayed
  – So is everyone else
  – But delays are common & unpredictable
The Fable drags on ... 

• Bob and Alice still have issues
The Fable drags on ...

• Bob and Alice still have issues
• So they need to communicate
The Fable drags on ...

• Bob and Alice still have issues
• So they need to communicate
• So they agree to use billboards ...
Billboards are Large
Write One Letter at a Time ...
To post a message

WASH THE CAR
Let’s send another message

SELL LAVA LAMPS
Uh-Oh
Readers/Writers

• Devise a protocol so that
  - Writer writes one letter at a time
  - Reader reads one letter at a time
  - Reader sees
    • Old message or new message
    • No mixed messages
Readers/Writers (continued)

• Easy with mutual exclusion
• But mutual exclusion requires waiting
  – One waits for the other
  – Everyone executes sequentially
• Remarkably
  – We can solve R/W without mutual exclusion
Why do we care?

• We want as much of the code as possible to execute concurrently (in parallel)
• A larger sequential part implies reduced performance
• Amdahl’s law: this relation is not linear...
Amdahl’s Law

\[
\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}
\]

...of computation given \( n \) CPUs instead of 1
Amdahl’s Law

\[ \text{Speedup} = \frac{1}{1 - p + \frac{p}{n}} \]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

Sequential fraction

Parallel fraction
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

- Sequential fraction
- Parallel fraction
- Number of processors
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}
\]
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?
Example

• Ten processors
• 80% concurrent, 20% sequential
• How close to 10-fold speedup?

\[
\text{Speedup} = \frac{1}{1 - 0.8 + \frac{0.8}{10}} = 3.57
\]
Example

• Ten processors
• 90% concurrent, 10% sequential
• How close to 10-fold speedup?
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}
\]
The Moral

• Making good use of our multiple processors (cores) means
• Finding ways to effectively parallelize our code
  – Minimize sequential parts
  – Reduce idle time in which threads wait without
Multicore Programming

• This is what this course is about...
  – The % that is not easy to make concurrent yet may have a large impact on overall speedup

• Next week:
  – A more serious look at mutual exclusion
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

• **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work

• **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

• For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - [http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/).

• Any of the above conditions can be waived if you get permission from the copyright holder.

• Nothing in this license impairs or restricts the author's moral rights.