Reasoning About Parallel Program

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07
Most of this lecture is based on slides from Maurice Herlihy and Nir Shavit
Motivation

• Shared memory programs can be complicated
  • Race conditions can produce unpredictable behavior
  • Can be avoided with locking, but that can be expensive

• Need to be more clever in some cases, especially with low level programming
  • Use locks only when necessary
  • May use (gasp!) shared memory operations if you really know what you’re doing

• How do we think about these kinds of programs and reason about their correctness?
Agenda

• Example to motivate problem
• Concurrency models
  • Partial orders, state machines,
• Correctness conditions
  • Serializability, …
  • Safety and liveness
• Clock synchronization
• Bag of tricks for parallel and distributed algorithm’s
  • Timestamps, markers,
Model of Concurrent Computation

threads

memory

object

object

© 2006 Herlihy and Shavit
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

• Larger numbers imply 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 $i$

```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
Where Things Reside

```java
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);
    }
}

Stop when every value taken
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++;
    }
}
```
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
public class Counter {
    private long value;

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

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

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

• Real multiprocessor problem
  • 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 …

• Real World interpretation
  • Message-passing doesn’t work
  • Recipient might not be
    • Listening or there at all
  • Communication must be
    • Persistent (like writing), not transient (like speaking)
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

• Real world 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
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

- 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

Alice’s last look

Bob’s last looked

QED

Alice must have seen Bob’s Flag. A Contradiction
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
• Leading to a new coordination problem: Producer-Consumer
Bob Puts Food in the Pond
Alice releases her pets to Feed

mmm...
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

© 2006 Herlihy and Shavit
Alice Resets Can when Pets are Fed
while (true) {
    while (can.isUp()) {};
    pet.release();
    pet.recapture();
    can.reset();
}
Pseudocode

Alice's code

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

Bob's code

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
Waiting

• Both solutions use waiting
  - `while(mumble) {}`

• Waiting is *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

Letter Tiles
From Scrabble™ box
Write One Letter at a Time …
To post a message

WASH THE CAR

whew
Let’s send another message
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
Approach

• Reasoning about a concurrent system:
  • Program imposes a order events in the system
  • Which event is known to happy before another (some may be unordered, since ordering is only partial)
  • Examples of events: Messages, memory operations, …

• Consider all possible topological sorts to serialize
• Each of those serial “histories” must be “correct.”
Happens Before Relation

• System is a set of processes
• Each process is a sequence of events (a total ordering) on events
• **Happens before**, denoted ->, is defined as the smallest relation s.t.:
  • 1) if a and b are in the same process, and a is before b, then a -> b
  • 2) If a is a message send and b is the matching receive, then a -> b
  • 3) if a->b and b->c then a->c (transitive)
Happens Before Example

- What if processes are multithreaded?
- How to we determine the events?
  - Send message, receive message, what else?
  - What about read/write? Nonblocking opns?
- Does this help if we’re trying to reason about a program or an algorithm?

© 2006 Herlihy and Shavit
Common Notions of Correctness

- Serializability, strong serializability
- Sequential consistency, total store ordering, weak ordering, …
- Linearizability (used in wait-free)

- All of these are based on the idea that operations (transactions) must appear to happen in some serial order
- Which of these (or others) are useful?
Variations on Correctness

• Why are there so many notions?
  • Do all processes observe the same “serial” order, or can some see different views?
  • Are the specifications of each operation deterministic? Can processes see different “correct” behaviors?
  • Are all operations executed by a process ordered? E.g., read x -> read y doesn’t matter.
  • Is the observed order consistent with real time?
Closed World Assumption

• Most of these correctness ideas assume that all communication is part of the system.

• Anomalies come from:
  • Phone calls between users
  • A second “external” network

• How does one prevent these anomalies?