Overview of Concurrency and Program Analysis

Dr. Jeff Huang
Fall 2014
Outline

• What is Concurrency?
• Why Concurrency?
• How to Program?
• Hard. Why?
• Basic Concepts
• Static Analysis & Dynamic Analysis
• Important Directions
What is Concurrency?

- Traditionally, the expression of a task in the form of multiple, possibly interacting subtasks, that may potentially be executed at the same time.
What is Concurrency?

• Concurrency is a programming concept.
• It says nothing about how the subtasks are actually executed.
• Concurrent tasks may be executed serially or in parallel depending upon the underlying physical resources available.
The modern world is parallel

- Multicore
- Networks
- Clouds of CPUs
- Loads of Users

- Capture the logical structure of certain applications
- Performance through use of multiple processors
Why Concurrency

• In a serial environment, consider the following simple example of a server, serving requests from clients (e.g., a web server and web clients)

![Diagram showing a serial server serving requests](image)
Why Concurrency

• Let’s process requests serially

Total completion time = 8 units, Average service time = (6 + 8)/2 = 7 units
Why Concurrency

• Why a concurrent server now!
Why Concurrency

• We reduced mean service time!

![Diagram showing time intervals and average service time calculation]

Total completion time = 8 units, Average service time = (4 + 8)/2 = 6 units
Why Concurrency

• The lesson from the example is quite simple:
  – Not knowing anything about execution times, we can reduce average service time for requests by processing them concurrently!

• But what if I knew the service time for each request?
  – Would “shortest job first” not minimize average service time anyway?
  – Aha! But what about the poor guy standing at the back never getting any service (starvation/ fairness)?
Why Concurrency

• Notions of service time, starvation, and fairness motivate the use of concurrency in virtually all aspects of computing:
  – Operating systems are multitasking
  – Web/database services handle multiple concurrent requests
  – Browsers are concurrent
  – Virtually all user interfaces are concurrent
Why Concurrency

• In a parallel context, the motivations for concurrency are more obvious:
  – Concurrency + parallel execution = performance
How to Program

• Abstractions
  – Shared memory, message-passing, data parallel
    • Erlang, MPI, Concurrent ML, Cuda
    • Posix, Cilk, OpenMP
  – Synchronous vs. asynchronous communication

• Data Structures and Algorithms
  – Queues, Heaps, Trees, Lists
  – Sorting, Graph Algorithms

• Processor Architectures
  – Relaxed memory models
  – GPGPU
How to Program

- Standard models of parallelism
  - Shared memory (Pthreads)
  - Message passing (MPI)
  - Data parallel (HPF, Chapel, X10)
  - Shared memory + data parallel (OpenMP)
Shared Memory

• Threads communicate by reading and writing to shared memory
  – Easier transition from sequential programs
  – Don’t have to construct new communication abstractions
• But, implicit communication via shared-memory raises complex issues of its own
  – Data races: concurrent (unintended) access to the same memory location
• How do we express concurrency and synchronization?
  – As language primitives (Java, C#, Cilk, X10...)
  – As library calls (Posix (Pthreads), Intel TBB)
Message Passing

- Processes communicate with each other without resorting to shared variables
- If P and Q wish to communicate, they need to
  - establish a communication link between them
  - exchange messages via send/receive
MPI (Message Passing Interface)

• Is the de facto message passing standard
• A library of routines (i.e. API, called in C/C++/Fortune/Java/C#/Python):
  – Provide message passing facilities (send and receive, in various flavors) to distribute and communicate data
  – Provide additional synchronization facilities
• Available on virtually all platforms, including public domain versions (MPICH, OpenMPI)
Data Parallel

• Partitioned Global Address Space (PGAS) model
  – Address space is treated globally
  – Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  – A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  – Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
Why Hard

• The human mind tends to be sequential
• The notion of time is often misleading
• Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
Why Hard

• Races: outcome depends on arbitrary scheduling decisions elsewhere in the system
  – Example: who gets the last seat on the airplane?
• Deadlock: improper resource allocation prevents forward progress
  – Example: traffic gridlock
• Livelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress
  – Example: people always jump in front of you in line
Thread Safety

• Suppose a program creates N threads, each of which calls the same procedure found in some library

• Suppose the library modifies some global (shared) data structure

• Concurrent modifications to this structure may lead to data corruption
Basic Concepts

• Concurrency bug patterns
  – Data race vs race conditions
  – Order violation, atomic Violations
  – Serializability, linearizability
  – Deadlocks

• Memory Models

• Program Analysis techniques
Data Race

• Every thread can observe actions of other threads on non-thread local data (e.g. heap)
• Data visible to multiple threads must be protected (synchronized) to ensure the absence of data races
  – A data race consists of two concurrent accesses to the same shared data by two separate threads, at least one of which is a write
Data Race

THREAD 1
a = data;
a++;
data += a;

THREAD 2
b = data;
b++;
data += b;

Assuming data = 0 initially, can data be 1 after the program completes?
Race Condition

- A flaw that occurs when the timing or ordering of events affects a program’s correctness.
- A semantic error!
- E.g., some kind of external timing or ordering non-determinism: context switches, OS signals, memory operations on a multiprocessor, and hardware interrupts.
Race Condition

Thread 1    Thread 2

lock(l)     lock(l)
x=1         x=2
unlock(l)   unlock(l)

class Race {
    static volatile int i;
    static int uniqueInt() { return i++; }
}
An order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 should not read mThread until thread 1 initializes mThread.

Note that, this bug could not be fixed by making PR CreateThread atomic with the write to mThread.
Atomicity Violation

• A series of related operations are interfered by certain other operations, which causes inconsistency.

A real bug in Mozilla. When Thread 2 violates the atomicity of Thread 1’s accesses to gCurrentScript, the program crashes.
Serializability

• Properties of parallel data structures
• A data structure with operations is serializable if
  – the state of any possible parallel execution, including
    • arguments to operations
    • return values from operations
    • internal state of data structure
  – is equivalent to some serialization of the operations applied
• In other words
  – there must exist an ordering of operations
  – such that the operations executed in parallel appear to have executed serially in that order
Serializability

• Conflict-serializability
  – Two executions are conflict-equivalent if and only if they contain the same events, and for each pair of conflicting events, the two events appear in the same order.

• View-serializability
  – Two executions are view-equivalent if they contain the same events. If each read operation reads the same result of the same write operation in both executions, and both execution must have the same final write for any location.
Serializability Analysis

Consider three accesses to a shared variable, the first and the third are by the same thread, the second by another thread.

<table>
<thead>
<tr>
<th>Case</th>
<th>Interleaving</th>
<th>Serializable?</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read-Read-Read</td>
<td>Y</td>
<td>Equivalent to Read-Read-Read</td>
</tr>
<tr>
<td>1</td>
<td>Write-Read-Read</td>
<td>Y</td>
<td>Equivalent to Write-Read-Read</td>
</tr>
<tr>
<td>2</td>
<td>Read-Write-Read</td>
<td>N</td>
<td>The Write may make two Reads return different values</td>
</tr>
<tr>
<td>3</td>
<td>Write-Write-Read</td>
<td>N</td>
<td>The Read does not get the local result it expects</td>
</tr>
<tr>
<td>4</td>
<td>Read-Read-Write</td>
<td>Y</td>
<td>Equivalent to Read-Read-Write</td>
</tr>
<tr>
<td>5</td>
<td>Write-Read-Write</td>
<td>N</td>
<td>The Read returns intermediate result</td>
</tr>
<tr>
<td>6</td>
<td>Read-Write-Write</td>
<td>N</td>
<td>The Write value is lost</td>
</tr>
<tr>
<td>7</td>
<td>Write-Write-Write</td>
<td>Y</td>
<td>Equivalent to Write-Write-Write</td>
</tr>
</tbody>
</table>
Atomic-set serializability

Consider accesses to multiple related variables?

Class Account{
    int checking, saving;
    transfer(int n) {
        checking-=n;
        saving+=n;
    }
}
Linearizability

• Each method call should appear to “take effect” instantaneously at some moment between its invocation and response.

• Two important properties:
  • local property: a system is linearizable iff each individual object is linearizable. It gives us composability.
  • non-blocking property: one method is never forced to wait to synchronize with another.
Deadlocks

Traffic jam

Dinning philosophers
Deadlock

- Program hangs due to missing notification

Subject.run() {
    ...
    synchronized (events) {
        events.add(...);
        events.notify();
    }
    ...
}

Observer.run() {
    ...
    synchronized (events) {
        events.wait();
        ...
        = events.get(0);
    }
    ...
}
Memory Model

• What values a read can return?

Initially $X == Y == 0$

Thread 1
$X = 1;$
$r1 = Y;$

Thread 2
$Y = 1;$
$r2 = X;$

Can $r1 == r2 == 0$? YES
Memory Model

• Programmers reason with interleaved semantics
• But hardware and compilers do not
  – They use sophisticated techniques for optimization
  – These techniques can “reorder” statements
Memory Model

• Sequential Consistency (SC)
  – Operations of a single thread in program order
  – All operations in a total order

• SC prohibits hardware performance
• Compiler transformation violates SC
Memory Model

• Data-race-freedom (DRF) model
  – SC for data-race-free programs
  – No guarantee for racy programs

<table>
<thead>
<tr>
<th>Initially X == Y == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>r1 = X;</td>
</tr>
<tr>
<td>Y = r1;</td>
</tr>
<tr>
<td>Thread 2</td>
</tr>
<tr>
<td>r2 = Y;</td>
</tr>
<tr>
<td>X = r2;</td>
</tr>
</tbody>
</table>

Can r1 == r2 == 42?  YES!

We must not allow it!
Memory Model

• Java Memory Model (JMM)
  – DRF + weak semantic for racy programs
  – Forbid out-of-thin-air

• C++0x
• x86_TSO
• C++ 11/14
• Goal: automatically detect/debug/fix concurrency errors, and test concurrent programs
# Program Analysis

- **Static analysis vs dynamic analysis:**

<table>
<thead>
<tr>
<th>Static analysis</th>
<th>Dynamic analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyze static code</td>
<td>Analyze running program</td>
</tr>
<tr>
<td>Find problems in any execution</td>
<td>Find problems in some real execution</td>
</tr>
<tr>
<td>Sound: no false negatives*</td>
<td>Unsound: false negatives</td>
</tr>
<tr>
<td>Imprecise: false positives</td>
<td>Precise: often no false positives</td>
</tr>
<tr>
<td>Doesn’t slow running program</td>
<td>Slows executing program</td>
</tr>
</tbody>
</table>
Control Flow Graph

\[
x := a + b; \\
y := a * b; \\
\textbf{while} (y > a) \{ \\
    a := a + 1; \\
    x := a + b \\
\}\]
Dataflow Analysis

• Live analysis: Is “x” alive at point p?
• Reaching definitions
• Available expressions
• Busy expressions
• ...

```
x := a + b;
y := a * b;
```

```
y > a
```

```
a := a + 1;
x := a + b
```
Dataflow Analysis

At a given program point $p$, which locals $v$ will be accessed in the future?

```java
void foo(boolean b) {
    int x = 5, y = 2; // {x, y, b}
    System.out.println(x); // {x, y, b}
    if (b) {
        x = bar(y*2); // {x}
    } else {
        foo(false); // {x}
    }
    System.out.println(x); // {}
}
```
Pointer Analysis

• What memory locations can a pointer expression refer to?
• E.g., int x; p = &x; q = p;

• A hard problem, but is essential for many compiler optimizations

• Alias analysis: When do two pointer expressions refer to the same storage location?
  – *p and *q alias, as do x and *p, and x and *q
Call Graph

```java
foo() {
    A o = ...;
    o.bar();
}

class A {
    bar() {
        /* */
    }
}
class B extends A {
    bar() {
        /* */
    }
}

"Which methods might o.bar() reach?"
Dynamic Profiling

long totalObjs = 0;
void main(String[] s){
    A a = new A(...);
    totalObjs ++;
    for(...){
        B b = new B(...);
        totalObjs ++;
    }
    print(totalObjs);
}

• A run-time technique that gathers execution information
  – E.g., total # objects created during the execution

• How--- via program instrumentation
Dynamic Slicing

• Record all memory accesses and their dependence relationships

• Applications
  – Automated debugging
  – Performance analysis

```java
void main(String[] s){
    a.f = ...;
    ...
    c = b.f;
}
```
Context Profiling

- Record calling contexts for method invocations

```java
void m(...){
    bar(new A()); //call 1
}
void n(...){
    bar (new B()); //call 2
}
void bar(A a){
    a.f = ...;
}
```
Context Profiling

- Record calling contexts for method invocations

- Applications
  - Context-sensitive dynamic analysis
  - Interprocedural compiler optimization

```java
void m(...){
    bar(new A()); //call 1
}
void n(...){
    bar (new B()); //call 2
}
void bar(A a){
    a.f = ...;
}
```
Symbolic Execution

- Use symbolic values, instead of concrete data values as input
- Represent the values of program variables as symbolic expressions over the symbolic input values.

- Each symbolic execution path stands for many actually program runs
- Thus can cover a lot more of the program’s execution space than testing can
Symbolic Execution

1. int a = \alpha, b = \beta, c = \gamma;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10. }
11. assert(x+y+z!=3)
Model Checking

Program / Model

void add(Object o) {
    buffer[head] = o;
    head = (head+1) % size;
}

Object take() {
    ... tail = (tail+1) % size;
    return buffer[tail];
}

Testing / Simulation

Test Oracle

Program / Model

void add(Object o) {
    buffer[head] = o;
    head = (head+1) % size;
}

Object take() {
    ... tail = (tail+1) % size;
    return buffer[tail];
}

Model Checking

Model Checking

Property

always(φ or ψ)
Important Directions

• Deterministic Multithreading
• Deterministic-by-default programming
• Transactional Memory
Deterministic Multithreading

• How to make concurrent program executions deterministic?
  – In the presence of data races
  – Run as fast as possible
  – Applicable to all multithreaded programs

• The DMP paper [ASPLOS 2009]: 200 citations

• Compiler/Library/Runtime:
  – DMP, Kendo, Dthread, Parrot, Grace, CoreDet, dOS, Determinator, RFDet ...
Deterministic-by-default Programming

• Design a language and type system to ensure the resulting program is always deterministic

• How expressive it can be?
• How fast?

• The DPJ paper [OOPSLA 2009]: 270 citations
Transactional Memory

• No lock-based synchronization!
• Simplify concurrent programming by grouping operations and running them like a single database transaction
  
  \[
  \text{atomic} \{
  \begin{align*}
  &\text{access object 1;} \\
  &\text{access object 2;}
  \end{align*}
  \]

• Upon exit of a transaction
  
  – If no other thread concurrently modified the same data, all of our changes will simultaneously be visible
  – Otherwise, discards all changes and roll back
Transactional Memory

- Software: TinySTM, DSTM2, ...
- Hardware: Intel Haswell, IBM Blue Gene/Q

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast (due to hardware operations)</td>
<td>Slow (due to software validation/commit)</td>
</tr>
<tr>
<td>Light code instrumentation</td>
<td>Heavy code instrumentation</td>
</tr>
<tr>
<td>HW buffers keep amount of metadata low</td>
<td>Lots of metadata</td>
</tr>
<tr>
<td>No need of a middleware</td>
<td>Runtime library needed</td>
</tr>
<tr>
<td>Only short transactions allowed (why?)</td>
<td>Large transactions possible</td>
</tr>
</tbody>
</table>
Specific Domains

- Event-driven systems (e.g., Android, browsers)
- Real-time embedded systems (e.g., cars, aircrafts, medical systems)
- GPGPU computing (CUDA, OpenCL)