Programming Language Theory

Concurrency and Shared Memory
Concurrency and Parallelism

- PL support for concurrency/parallelism a huge topic
  - Increasingly important
  - Lots of active research in the area

- We will just focus on explicit threads plus:
  - Shared memory (locks and transactions)
  - Futures
  - Synchronous message passing (Concurrent ML)

- We’ll skip
  - Process calculi (foundational message-passing)
  - Asynchronous methods, join calculus, …
  - Data-parallel languages (NESL)
  - …

- Mostly in ML syntax (inference rules where convenient)
Concurrency vs. Parallelism

One take on this (terminology not universal, but distinction important):

Software is **concurrent** if a primary intellectual challenge is responding to external events from multiple sources in a timely manner.

- Examples: operating system, GUI, version control
- Key challenge is responsiveness
- Often provide responsiveness via threads
- Inherently non-deterministic, but not necessarily parallel

Software is **parallel** if a primary intellectual challenge is using extra computational resources to do more useful work per unit time.

- Examples: scientific computing, most graphics, a lot of servers
- Key challenge is Amdahl’s Law (no sequential bottlenecks)
- Often provide parallelism via threads on different processors
- Ideally deterministic, but not when concurrent
Threads

High-level: “Communicating sequential processes”
Low-level: “Multiple stacks plus communication”
Threads

High-level: “Communicating sequential processes”
Low-level: “Multiple stacks plus communication”

From SML/NJ’s structure CML:

(* thread handle *)
type thread_id
(* create a new thread of control *)
val spawn : (unit -> unit) -> thread_id
(* thread id of calling thread *)
val getTid : unit -> thread_id

The code for a thread is in a closure (with hidden fields) and spawn actually executes the thread.
Threads

High-level: “Communicating sequential processes”
Low-level: “Multiple stacks plus communication”

From SML/NJ’s structure CML:

(* thread handle *)

**type** thread_id

(* create a new thread of control *)

**val** spawn : (unit -> unit) -> thread_id

(* thread id of calling thread *)

**val** getTid : unit -> thread_id

The **code** for a thread is in a closure (with hidden fields) and **spawn** actually **executes** the thread.

Most languages make the same distinction, e.g., Java:

- Create a **Thread** object (just the code and data)
- Call its **start** method to actually execute the thread
Why use threads?

One or more of:

1. Performance (multiprocessor or mask I/O latency)
2. Isolation (separate errors or responsiveness)
3. Natural code structure (1 stack awkward)

It’s not just performance.
Formalizing threads

- Machine state is one heap and multiple expressions ("thread pool")
- For the machine’s “take a step”, any $e_i$ might “take a step”
  - And $e_i$ might spawn a new thread
- Any $e_i$ that is a value can not take a step, and can be removed from the “thread pool”
- Nondeterministic with *interleaving granularity* determined by granularity of evaluation rules
Formalizing threads

\[
\begin{align*}
e &::= \cdots \mid \text{spawn}(e) \\
v &::= \cdots \\
o &::= \cdot \mid e \\
H &::= \cdot \mid H, a \mapsto v \\
T &::= e_1, \ldots, e_n
\end{align*}
\]

\[H; T \rightarrow H'; T'\]

\[H; e \rightarrow H'; e'; o'\]
Formalizing threads (continued)

\[ H; T \rightarrow H'; T' \]

\[ H; e \rightarrow H'; e'; \cdot \]
\[ H; \cdot, e_1, \ldots, e, \ldots, e_n \rightarrow H'; \cdot, e_1, \ldots, e', \ldots, e_n \]

\[ H; e \rightarrow H'; e'; e'' \]
\[ H'; \cdot, e_1, \ldots, e, \ldots, e_n \rightarrow H'; \cdot, e_1, \ldots, e', \ldots, e_n, e'' \]

\[ H; \cdot, e_1, \ldots, e_{i-1}, v, e_{i+1}, \ldots, e_n \rightarrow H; \cdot, e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \]

Program termination: \( H; \cdot \)
Formalizing threads (continued)

\[ H; e \rightarrow H'; e'; o' \]

\[ \frac{H; e_f \rightarrow H'; e'_f; o'}{H; e_f e_a \rightarrow H'; e'_f e_a; o'} \quad \frac{H; e_a \rightarrow H'; e'_a; o'}{H; v_f e_a \rightarrow H'; v_f e'_a; o'} \]

\[ \frac{H; (\lambda x. e_b) v_a \rightarrow H; e_b[v_a/x]; \cdot}{H; e_r \rightarrow_{cbv} H'; e'_r; o'} \]

\[ \frac{H; ! e_r \rightarrow H'; ! e'_r; o'}{H(a) = v \quad H; ! a \rightarrow H; v; \cdot} \]

\[ H; e_f \rightarrow H'; e'_f; o' \]

\[ H; \text{spawn}(e_f) \rightarrow H'; \text{spawn}(e'_f); o' \]

\[ H; \text{spawn}(v_f) \rightarrow H; (); v_f () \]
Equivalence just changed

Expressions equivalent in a single-threaded world are not necessarily equivalent in a multithreaded context!

Example in SML:

```sml
val (x, y) = (ref 0, ref 0)
val _ = spawn (fn () => if (!y)=1 then x:=(!x)+1 else ())
val _ = spawn (fn () => if (!x)=1 then y:=(!y)+1 else ()) (* 1 *)
```

Can we replace line (1) with:

```sml
val _ = spawn (fn () => y:=(!y)+1; if (!x)<>1 then y:=(!y)-1 else ())
```

For more compiler gotchas, read "Threads cannot be implemented as a library" by Hans-J. Boehm (PLDI 05)

▶ Example: C bit-fields or other adjacent fields
Equivalence just changed

Expressions equivalent in a single-threaded world are not necessarily equivalent in a multithreaded context!

Example in SML:

```sml
val (x, y) = (ref 0, ref 0)
val _ = spawn (fn () => if (!y)=1 then x:=(!x)+1 else ())
val _ = spawn (fn () => if (!x)=1 then y:=(!y)+1 else ()) (* 1 *)
```

Can we replace line (1) with:

```sml
val _ = spawn (fn () => y:=(!y)+1; if (!x)<1 then y:=(!y)-1 else ())
```
Equivalence just changed

Expressions equivalent in a single-threaded world
are not necessarily equivalent in a multithreaded context!

Example in SML:

```sml
val (x, y) = (ref 0, ref 0)
val _ = spawn (fn () => if (!y)=1 then x:=(!x)+1 else ()
val _ = spawn (fn () => if (!x)=1 then y:=(!y)+1 else ()) (* 1 *)
```

Can we replace line (1) with:

```sml
val _ = spawn (fn () => y:=(!y)+1; if (!x)<>1 then y:=(!y)-1 else ()
```

For more compiler gotchas, read
“Threads cannot be implemented as a library” by Hans-J. Boehm (PLDI 05)

Example: C bit-fields or other adjacent fields
Communication

If threads do nothing other threads need to “see”, then we are done

▶ best to do as little communication as possible
▶ e.g., do not mutate shared data unnecessarily, or hide mutation behind easier-to-use interfaces

One way to communicate: Shared memory

▶ One thread writes to a ref, another reads it
▶ Sounds (and is) nasty with pre-emptive scheduling
  ▶ Even with non-pre-emptive scheduling, hard to reason about
▶ Hence synchronization mechanisms
  ▶ Taught in O/S for historical reasons!
  ▶ Fundamentally about restricting interleavings
Fork-Join

“Fork-Join” parallelism a simple approach good for:
“farm out subcomputations then merge results”.

Common pattern:

```ocaml
val fork_join : ('a -> 'b array) -> ('b -> 'c) -> ('c array -> 'd) -> 'a -> 'd
```

Apply the second argument to each element of the 'b array in parallel,
then use third argument after they are done.

(* suspend caller until arg thread terminates *)

```ocaml
val join : thread_id -> unit
```
Fork-Join vs. Map-Reduce

“Map-Reduce” parallelism a simple approach good for:
“farm out subcomputations then merge results”.

Common pattern:

```ocaml
val map_reduce : ('a -> 'b array) -> (* divider *)
               ('b -> 'c) -> (* mapper *)
               ('c array -> 'd) -> (* reducer *)
               'a ->
               'd (* result *)
```

Apply the second argument to each element of the 'b array in parallel,
then use third argument after they are done.
Fork-Join vs. Map-Reduce

“Map-Reduce” parallelism a simple approach good for:
“farm out subcomputations then merge results”.

Common pattern:

```plaintext
val map_reduce : ('a -> 'b array) -> (* divider *)
    ('b -> 'c) -> (* mapper *)
    ('c array -> 'd) -> (* reducer *)
    'a -> (* data *)
    'd (* result *)
```

Apply the second argument to each element of the 'b array in parallel, then use third argument after they are done.

I modified Google’s Map-Reduce (very) little to emphasize connection. Obviously, much more to work in a distributed and fault-tolerant setting.
Futures

A different model for explicit parallelism without using explicit shared memory or messages.

- Easy to implement on top of either, but most models are easily inter-implementable.

```ocaml
type 'a promise
(* execute in parallel *)
val future : (unit -> 'a) -> 'a promise
(* synchronize on result *)
val force : 'a promise -> 'a
```

Essentially fork/join with a value returned?

- Returning a value more functional
- Less structured than `cobegin s1; s2; ...; sn` form of fork/join
- No `performance` advantage in a non-parallel (but concurrent) setting.
Futures

type 'a promise = thread_id * 'a option ref

fun future thunk = 
    let val r = ref NONE 
    val tid = spawn (fn () => r := SOME (thunk ())) 
    in (tid, r) 
end

fun force (tid, r) = 
    let val _ = join tid 
    in case !r of 
        NONE => raise Fail "impossible"
        | SOME z => z 
    end
Locks (a.k.a. mutexes)

```ocaml
type mutex (* a mutex *)
val create : unit -> mutex
val lock : mutex -> unit (* may block *)
val try_lock : mutex -> bool (* does not block *)
val unlock : mutex -> unit
```

Some features of locks:

- Reentrancy (changes semantics of `lock`)
- Banning nonholder release (changes semantics of `unlock`)

Using locks

Among infinite correct idioms using locks (and more incorrect ones), the most common:

▶ Determine what data must be “kept in sync”
▶ Always acquire a lock before accessing that data and release it afterwards
▶ Have a partial order on all locks and if a thread holds \( m_1 \) it can acquire \( m_2 \) only if \( m_1 < m_2 \).

Coarser locking (more data with same lock) trades off parallelism with synchronization.
▶ Acquire fewer locks, but might spend more time waiting for locks.

( Related: Performance-bug of false sharing.)
null
Formalizing locks

- Machine state is a heap, a thread pool, and a set of held locks.
- For reentrancy and banned nonholder releases, record the thread id of lock holders.
Getting it wrong

Races result from too little synchronization

- Data races: simultaneous read-write or write-write of same data
- Lots of PL work in last 10 years on types and tools to prevent/detect
- Provided language has some guarantees, may not be a bug
  - Canonical example: parallel search and “done” bits

- Higher-level races: much tougher to prevent in the language
  - Amount of correct nondeterminism inherently application/problem specific

Deadlock results from too much synchronization

- Cycle of threads waiting for someone else to do something
- Easy to detect dynamically with locks, but then what?
The Evolution Problem

Write a new function that needs to update $o_1$ and $o_2$ together.

What locks should you acquire? In what order?

There may be no answer that avoids races and deadlocks without breaking old code. (Need a stricter partial order.)
fun xferRace1 a1 a2 f =
  (get a1 f; put a2 f)

fun xferRace2 a1 a2 f =
  (put a2 f; get a1 f)

(* breaks abstraction and doesn't work *)
fun xferDeadlock (a1 as {lk = lk1, bal = bal1, avail = avail1})
  (a2 as {lk = lk2, bal = bal2, avail = avail2})
  f =
    let val _ = Mutex.lock lk1
      val _ = Mutex.lock lk2
      val _ = if !avail1 > f
        then (bal1 := !bal1 - f;
               avail1 := !avail1 - f;
               bal2 := !bal2 + f;
               avail2 := !avail2 + (if f < 500.0 then f else 500.0))
        else ()
      val _ = Mutex.unlock lk2
      val _ = Mutex.unlock lk1
    in () end
The Evolution Problem

Write a new function that needs to update $o_1$ and $o_2$ together.

- What locks should you acquire? In what order?

There may be no answer that avoids races and deadlocks without breaking old code. (Need a stricter partial order.)

Real example from Java:

```java
synchronized append(StringBuffer sb) {
    int len = sb.length(); //synchronized call
    if(this.count+len > this.value.length) this.expand(...);
    sb.getChars(0,len,this.value,this.count); //synchronized call
    ...
}
```

Undocumented in 1.4; in 1.5 caller synchronizes on sb if necessary.
Software Transactions

One of the hottest areas in CS research right now.

Java: `atomic { s }`

SML: `val atomic : (unit -> 'a) -> 'a`

Execute the body/thunk *as though* no interleaving from other threads.
▶ Allow parallelism unless there are actual run-time memory conflicts
  ▶ (detect and abort/retry)
▶ Convenience of coarse-grained locking with parallelism of fine-grained locking
▶ But language implementation has to do more to detect conflicts
  ▶ (much like garbage collection is convenient but has costs)

Most research is on implementation:
▶ preserve parallelism unless there are conflicts
Transactions make things easier

Problems like `append` and `xfer` become trivial.

So does mixing coarse-grained and fine-grained operations
- (e.g., hashtable lookup and hashtable resize)

Transactions are great, but not a panacea:
- Application-level races can remain
- Application-level deadlock can remain
- Implementations generally try-and-abort, which is hard for “launch missiles” (e.g., I/O)
- Many software implementations provide a weaker and under-specified semantics
- Memory-consistency model questions remain and may be worse than with locks . . .
Formalizing transactions

- Machine state is a heap, a thread pool, and an (optional) atomic thread
- Operational semantics are deceptively (?) simple, but provides no parallelism
- Much of the real work is in finding an implementation that is equivalent to the operational semantics, but provides more parallelism
- Tough questions of when is an implementation equivalent to an operational semantics
Memory models

A *memory-consistency model* (or just *memory model*)
for a concurrent shared-memory language
specifies “which write a read can see”.

The gold standard is *sequential consistency* (Lamport):
“the results of any execution is the same as if the operations of all
the processors were executed in some sequential order, and the opera-
tions of each individual processor appear in this sequence in the order
specified by its program”

Under sequential consistency, this assert cannot fail, despite data races:

```plaintext
val (x, y) = (ref 0, ref 0)
val _ = spawn (fn () => (x := 1; y := 1))
val _ = spawn (fn () => (let val r = !y
val s = !x
in assert(s>=r) end)
```

Matthew Fluet
Programming Language Theory
Lecture 18 27
Relaxed memory models

Modern imperative and OO languages
do not promise sequential consistency
(if they say anything at all)

- The hardware makes it prohibitively expensive
- Renders unsound almost every compiler optimization
  Example: common-subexpression elimination.

Initially: $a == b == 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = a + b;$</td>
<td>$b = 1;$</td>
</tr>
<tr>
<td>$y = a;$</td>
<td>$a = 1;$</td>
</tr>
<tr>
<td>$z = a + b;$</td>
<td></td>
</tr>
<tr>
<td>assert ($z &gt;= y$);</td>
<td></td>
</tr>
</tbody>
</table>
Relaxed $\neq$ Nothing

But (especially in a safe language) have to promise something

- When is code “correctly synchronized”?  
- What can a compiler do in the presence of races?  
  - Cannot seg-fault Java or compromise the SecurityManager  
  - Can a race between $x:=1$ and $!x$  
    cause the latter to produce a value “out of thin air” (Java: no).

The definitions are very complicated  
and programmers can usually ignore them,  
but do not assume sequential consistency.

See also Java’s volatiles and C++ atomics.  
C will likely adopt the C++ work.
In real languages

Java:

*If* every sequentially consistent execution of program \( P \) is data-race free, *then* every execution of program \( P \) is equivalent to some sequentially consistent execution.

- Not the definition, a theorem about the definition.
- Actual definition very complicated, balancing needs of code writers, compiler optimizers, and hardware.
  - Not defined in terms of “list of acceptable optimizations”

C++ (proposed):

Roughly, any data race is as undefined as an array-bounds error. *No such thing as a benign data race* and **no** guarantees if you have one.
(In practice, programmers will still assume things, like they do with casts.)

- But same theorem as Java: “DRF \( \Rightarrow \) SC”

Most languages: Eerily silent.
Mostly functional wins again

If most of your data is immutable and most code is known to access only immutable data, then most code can be optimized without any concern for the memory model.

So can afford to be very conservative for the rest.

Example: An SML program that uses references only for shared-memory communication.

Non-example: A Java program that uses mutable memory for almost everything.

Compilers try to figure out what is *thread-local* (again avoids memory-model issues), but it’s not easy.
Ordering and atomic

Initially: $x = y = 0$

Thread 1
\[
x = 1; \\
y = 1;
\]

Thread 2
\[
r = y; \\
s = x;
\]

Can $s$ be less than $r$?

Matthew Fluet
Programming Language Theory
Lecture 18
Ordering and atomic

Initially: $x == y == 0$

Thread 1

$x = 1;$

$y = 1;$

Thread 2

$r = y;$

$s = x;$

Can $s$ be less than $r$?

Yes. (Because of relaxed memory model.)
Ordering and atomic

Initially: $x == y == 0$

Thread 1
- $x = 1$
- $y = 1$

Thread 2
- $r = y$
- $s = x$

Can $s$ be less than $r$?

In Java, no.
Ordering and atomic

Initially: $x = y = 0$

Thread 1
\[
\begin{align*}
x &= 1; \\
\text{sync}(lk) &{} \\
y &= 1; \\
\end{align*}
\]

Thread 2
\[
\begin{align*}
r &= y; \\
\text{sync}(lk) &{} \\
s &= x; \\
\end{align*}
\]

Can $s$ be less than $r$?

In Java, no.
Ordering and atomic

Initially: $x = y = 0$

Thread 1
$x = 1;$
atomic 
$y = 1;$

Thread 2
$r = y;$
atomic 
$s = x;$

Can $s$ be less than $r$?
Ordering and atomic

Initially: \( x = y = 0 \)

Thread 1
- \( x = 1; \)
- atomic {}  
- \( y = 1; \)

Thread 2
- \( r = y; \)
- atomic {}  
- \( s = x; \)

Can \( s \) be less than \( r \)?

No universal consensus, but, in practice, yes!