Concurrency

(and Parallelism)
Parallelism and Concurrency in 251

• Goal: encounter
  – essence, key concerns
  – non-sequential thinking
  – some high-level models
  – some mid-to-high-level mechanisms

• Non-goals:
  – performance engineering / measurement
  – deep programming proficiency
  – exhaustive survey of models and mechanisms
Parallelism

Use more resources to complete work faster.

Data / work divided among workers = resources

Concurrency

Coordinate access to shared resources.

Workers = computations share data = resources

Both can be expressed using a variety of primitives.
Concurrency via Concurrent ML

• Extends SML with language features for concurrency.
• Included in SML/NJ and Manticore
• Model:
  – explicitly threaded
  – message-passing over channels
  – first-class events
CML: spawn explicit threads

vs. Manticore's "hints" for implicit parallelism.

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

let fun f () = new thread's work...
   val t2 = spawn f

in
   this thread's work ...

end

Diagram:

Thread 1

spawn f

thread 1 continues

Thread 2

new thread runs f

time
(Aside: different model, fork-join)

**fork**: (unit \(\rightarrow\) 'a) \(\rightarrow\) 'a task
"call" a function in a new thread

**join**: 'a task \(\rightarrow\) 'a
wait for it to "return" a result

Mainly for explicit **task parallelism**, not concurrency.

(CML's threads are similar, but cooperation is different.)
CML: How do threads cooperate?

\[
\text{val spawn : (unit -> unit) -> thread_id}
\]

How do we pass values in?  How do we get results of work out?

\[
\begin{align*}
\text{let val data_in_env = ...} \\
\text{fun closures_for_the_win x = ...} \\
\text{val _ = spawn (fn () =>} \\
\text{map closures_for_the_win data_in_env)} \\
\end{align*}
\]
CML: How do threads cooperate?

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

Threads communicate by passing messages through channels.

type 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
Tiny channel example

val channel : unit -> 'a chan

let val ch : int chan = channel ()

fun inc () =
  let val n = recv ch
  val () = send (ch, n + 1)
  in exit () end

in
  spawn inc;
  send (ch, 3);
  ...
  recv ch
end
Concurrent streams

fun makeNatStream () =
  let val ch = channel ()
    fun count i =
      send (ch, i);
      count (i + 1)
  in
    spawn (fn () => count 0);
    ch
  end

fun sum stream 0 acc = acc
| sum stream n acc =
  sum stream (n - 1) (acc + recv stream)

val nats = makeNatStream ()
val sumFirst2 = sum nats 2 0
val sumNext2 = sum nats 2 0
A common pattern: looping thread

```ocaml
fun forever init f = 
  let
    fun loop s = loop (f s)
  in
    spawn (fn () => loop init);
    ()
  end
```
Concurrent streams

fun makeNatStream () =
  let
    val ch = channel ()
  in
    forever 0 (fn i => (send (ch, i);
                        i + 1));
    ch
  end

see cml-sieve.sml, cml-stream.sml
fun makeNatStream () =
   let val ch = channel ()
       fun count i = (
           send (ch, i);
           count (i + 1)
       )
   in
       spawn (fn () => count 0);
       ch
   end

val nats = makeNatStream ()
val _ =
    spawn (fn () => print (Int.toString (recv nats)))
val _ = print (Int.toString (recv nats))
Synchronous message passing (CML)

message passing = handshake
receive blocks until a message is sent
send blocks until the message received

vs

asynchronous message passing
receive blocks until a message has arrived
send can finish immediately without blocking
Synchronous message passing (CML)

Thread 1

- send (ch, 0)
- recv ch

blocked until another thread receives on ch.

Thread 2

- ch
- recv ch
- send (ch, 1)

blocked until another thread sends on ch.

Concurrency 15
Asynchronous message passing
(not CML)

send does not block

send (ch, 0) → recv ch

Thread 1

send (ch, 0)

send (ch, 0)

send (ch, 0) → recv ch

Thread 2

recv ch

recv ch

recv ch

blocked until a thread first sends on ch.

Concurrent
First-class events, combinators

Event constructors

val sendEvt : ('a chan * 'a) -> unit event
val recvEvt : 'a chan -> 'a event

Event combinators

val sync : 'a event -> 'a
val choose : 'a event list -> 'a event
val wrap : ('a event * ('a -> 'b)) -> 'b event

val select = sync o choose
Utilities

val recv = sync o recvEvt
val send = sync o sendEvt

fun forever init f =
  let
    fun loop s = loop (f s)
  in
    spawn (fn () => loop init);
  ()
end
Why combinators?

```ocaml
fun makeZipCh (inChA, inChB, outCh) =
  forever () (fn () =>
    let
      val (a, b) = select [
        wrap (recvEvt inChA,
          fn a => (a, recv inChB)),
        wrap (recvEvt inChB,
          fn b => (recv inChA, b))
      ]
    in
      send (outCh, (a, b))
    end)
```

Remember:
synchronous (blocking) message-passing
More CML

- Emulating mutable state via concurrency: `cml-cell.sml`
- Dataflow / pipeline computation: `cml-sieve.sml`
- Implement futures: `cml-futures.sml`
Why avoid mutation (of shared data)?

• For parallelism?
• For concurrency?

Other models:
  
  Shared-memory multithreading + synchronization

...
Shared-Memory Multithreading

**Shared:**
heap and globals

**Unshared:**
locals and control

Implicit communication through sharing.
Concurrency and Race Conditions

```java
int bal = 0;

Thread 1
  t1 = bal
  bal = t1 + 10

Thread 2
  t2 = bal
  bal = t2 - 10

bal == 0
```
Concurrency and Race Conditions

```c
int bal = 0;

Thread 1
  t1 = bal
  bal = t1 + 10

Thread 2
  t2 = bal
  bal = t2 - 10

bal == -10
```
Concurrent and Race Conditions

Lock m = new Lock();
int bal = 0;

Thread 1
synchronized(m) {
    t1 = bal
    bal = t1 + 10
}

Thread 2
synchronized(m) {
    t2 = bal
    bal = t2 - 10
}

Thread 1
Thread 2

acquire(m)
t2 = bal
bal = t2 - 10
release(m)

acquire(m)
t1 = bal
bal = t1 + 10
release(m)