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

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
Explicit threads: spawn
vs. Manticore's "hints" for implicit parallelism.

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

let \( \text{fun f () = new thread's work...} \)
val \( t2 = \text{spawn f} \)

in \( \text{this thread's work...} \)
end

Concurrency 5

Another thread/task model: fork-join

fork : (\text{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.)

Concurrency 6

CML: How do threads cooperate?

val spawn : (\text{unit} \rightarrow \text{unit}) \rightarrow \text{thread_id}

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

let \( \text{val data_in_env = ...} \)
fun \( \text{closures_for_the_win x = ...} \)
val \( _ = \text{spawn (fn () => map closures_for_the_win data_in_env)} \)

in ...
end

Concurrency 7

CML: How do threads cooperate?

val spawn : (\text{unit} \rightarrow \text{unit}) \rightarrow \text{thread_id}

How do we get results of work out?

Threads communicate by passing messages through channels.

\[
\text{type 'a chan}
\]
val \( \text{recv} : ('a chan) \rightarrow 'a \)
val \( \text{send} : ('a chan \times 'a) \rightarrow \text{unit} \)
Tiny channel example

```ml
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

```ml
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

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

Concurrent streams

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

see cml-sieve.sml, cml-stream.sml
Ordering?

```ml
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**

**Asynchronous** message-passing (not CML)

send does not **block**

blocked until another thread sends on ch.

Thread 1

- recv ch
- send (ch, 0)

Thread 2

- recv ch
- ch

Thread 1

- send (ch, 0)
- ch
- send (ch, 1)

Thread 2

- recv ch
- ch
- recv ch
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

More CML
• Emulating mutable state via concurrency: cml-cell.sml
• Dataflow / pipeline computation
• Implement futures

Why combinators?
fun makeZipCh (inChA, inChB, outCh) =
  forever () (fn () =>
    let
      val (a, b) = select [
        wrap (recvEvt inCh1,
          fn a => (a, recv inChA)),
        wrap (recvEvt inCh2,
          fn b => (recv inChB, b))
      ]
    in
      send (outCh, (a, b))
    end)
To be continued...