Concurrency
(and Parallelism)

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.

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
CML: spawn explicit threads

vs. Manticore’s “hints” for implicit parallelism.

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

let fun f () = new thread’s work...
    val t2 = spawn f
in  this thread’s work ...
end
```

(Aside: different model, fork-join)

```ml
fork : (unit -> 'a) -> 'a task
"call" a function in a new thread

join : 'a task -> '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?

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

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

```ml
let val data_in_env = ...
    fun closures_for_the_win x = ...
    val _ = spawn (fn () =>
        map closures_for_the_win data_in_env)
in ...
end
```

Threads communicate by passing messages through channels.

```ml

val recv : ‘a chan -> ‘a
val send : (‘a chan * ‘a) -> 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 ()
        val ch = channel ()
        fun count i = (send (ch, i);
                        count (i + 1))
    in
        forever 0 (fn i => (send (ch, i);
                              i + 1));
        ch
    end
```

see cml-sieve.sml, cml-stream.sml
Ordering of steps, events?

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

val nats = makeNatStream ()
val _ =
  spawn (fn () => print (Int.toString (recv nats)))
val _ = print (Int.toString (recv nats))

Synchronous message passing (CML)

Synchronous 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
recv ch

Thread 1
Thread 2

Thread 1
Thread 2

blocked until another thread receives on ch.
recv ch

blocked until another thread sends on ch.
send (ch, 0)

recv ch

send (ch, 1)

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv ch

recv 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

Why combinators?

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)

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...

Concurrent and Race Conditions

```java
int bal = 0;

Thread 1
  t1 = bal
  bal = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10

Thread 1
  t1 = bal
  bal = t1 + 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

Concurrent and Race Conditions

```java
int bal = 0;

Thread 1
  t1 = bal
  bal = t1 + 10

Thread 2
  t2 = bal
  bal = t2 - 10

Thread 1
  t1 = bal
  bal = t1 + 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```

```java
int bal = 0;

Thread 1
  t1 = bal
  t1 = t1 + 10
  t2 = bal
  bal = t2 - 10

Thread 2
  t2 = bal
  bal = t2 - 10
```
Concurrency 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
}

acquire(m)
release(m)

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
}

acquire(m)
release(m)