Concurrency

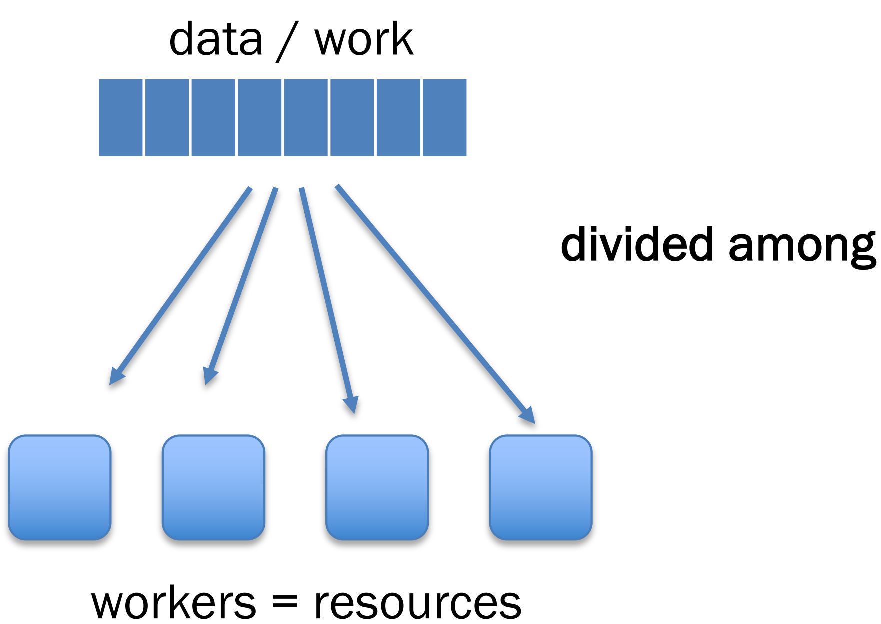
(and Parallelism)

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

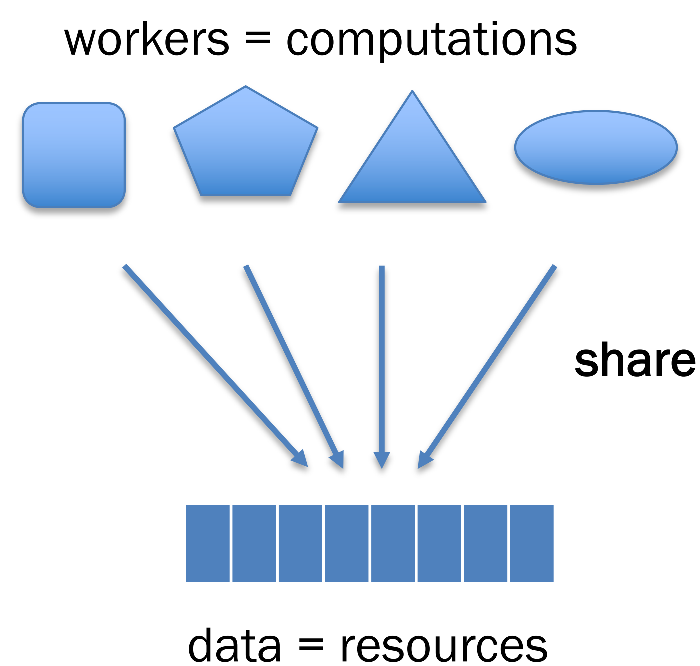
Parallelism

Use more resources
to complete work faster.



Concurrency

Coordinate access
to shared 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

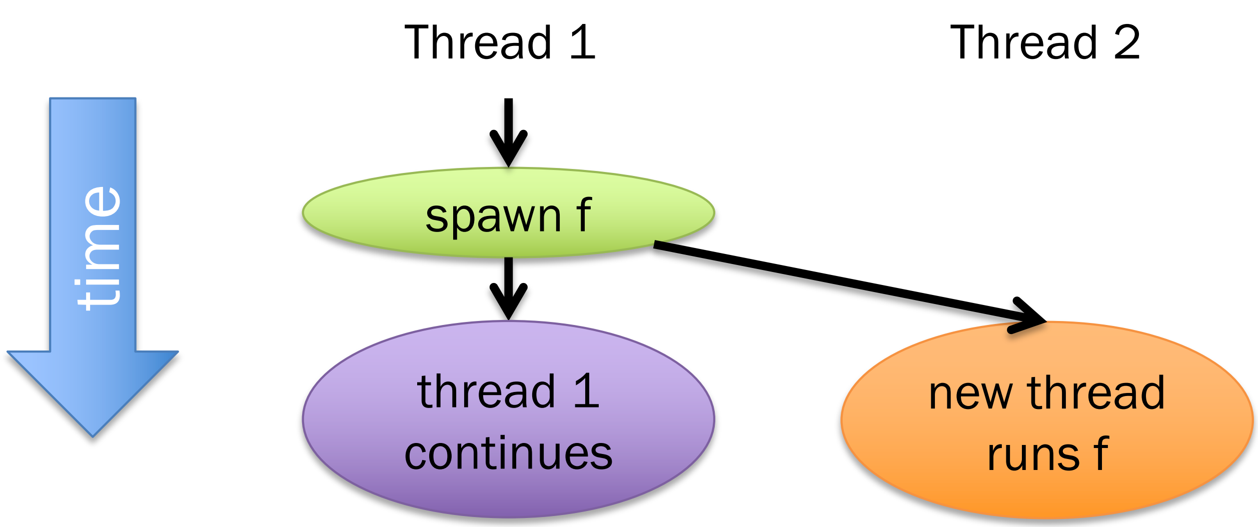
Explicit threads: `spawn`

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

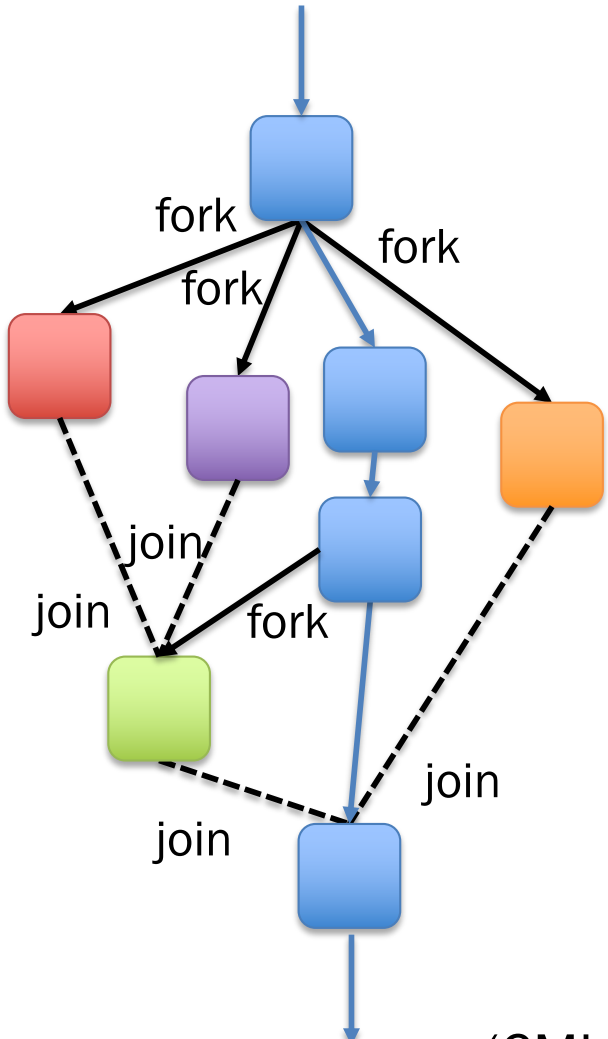
workload thunk

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

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



Another thread/task model: *fork-join*




```
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*?

workload thunk

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

How do we pass values in? ✓

How do we get results of work out?

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

CML: How do threads cooperate?

workload thunk

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

How do we get results of work out?

Threads **communicate** by passing messages through **channels.**

```
type 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

Draw time diagram.

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
```

Draw time diagram.

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

```
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

Draw time diagram.

Ordering?

```
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

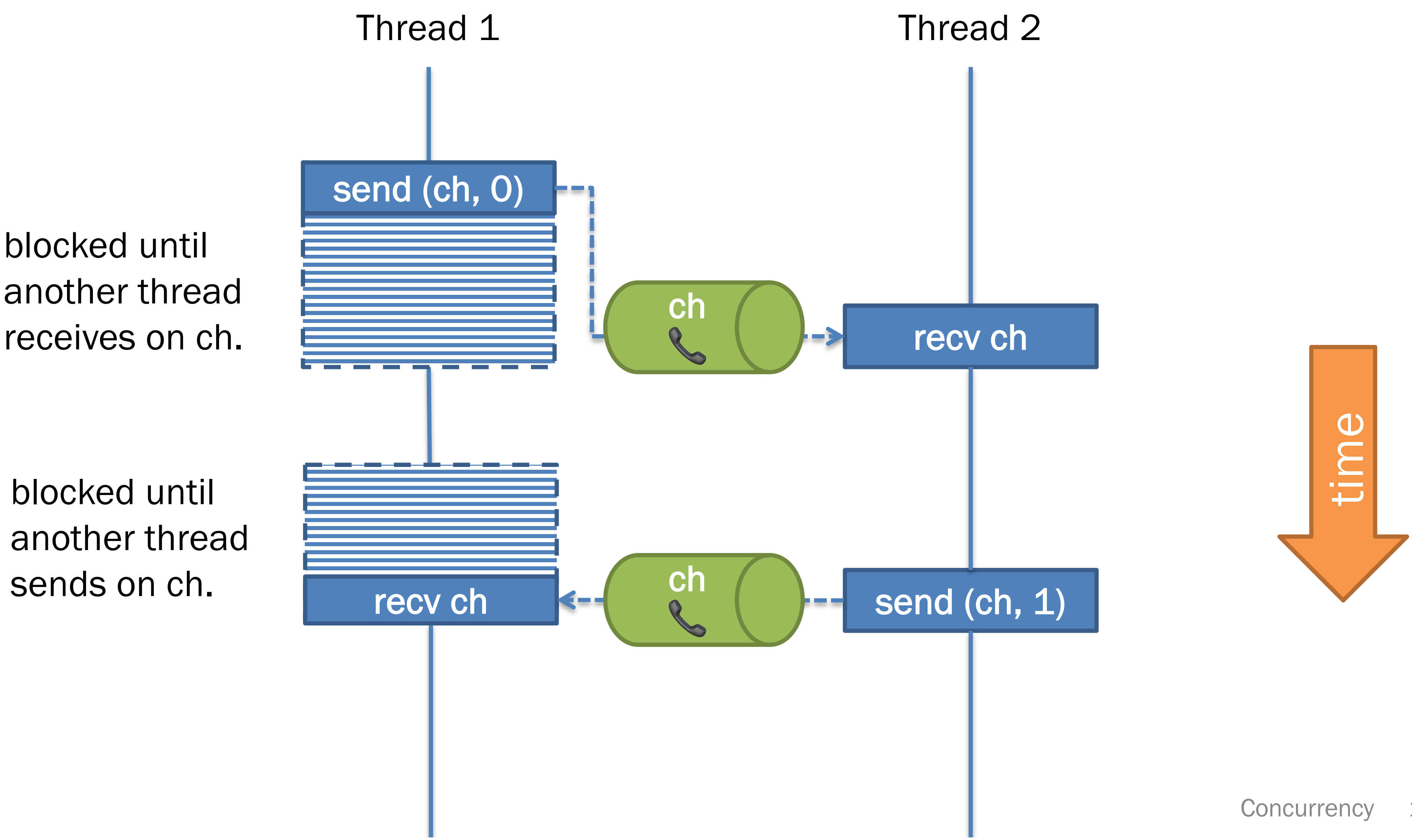
send *blocks* until the message received

vs 📬 *asynchronous* message-passing

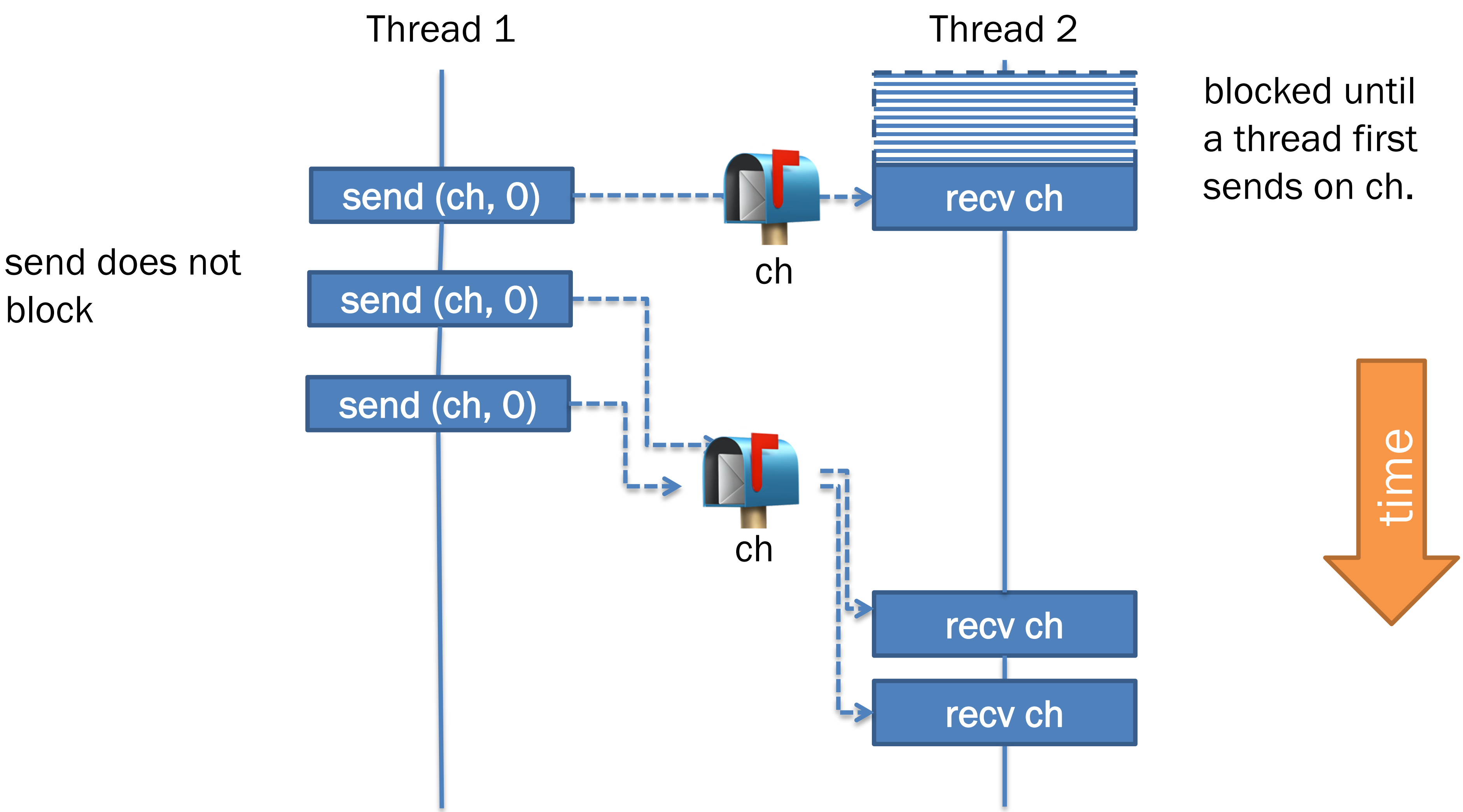
receive *blocks* until a message has arrived

send can finish immediately without blocking

Synchronous message-passing (CML)



Asynchronous message-passing (not CML)



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?

Remember:
synchronous (blocking)
message-passing

```
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
- Implement futures

Why avoid mutation?

- For parallelism?
- For concurrency?

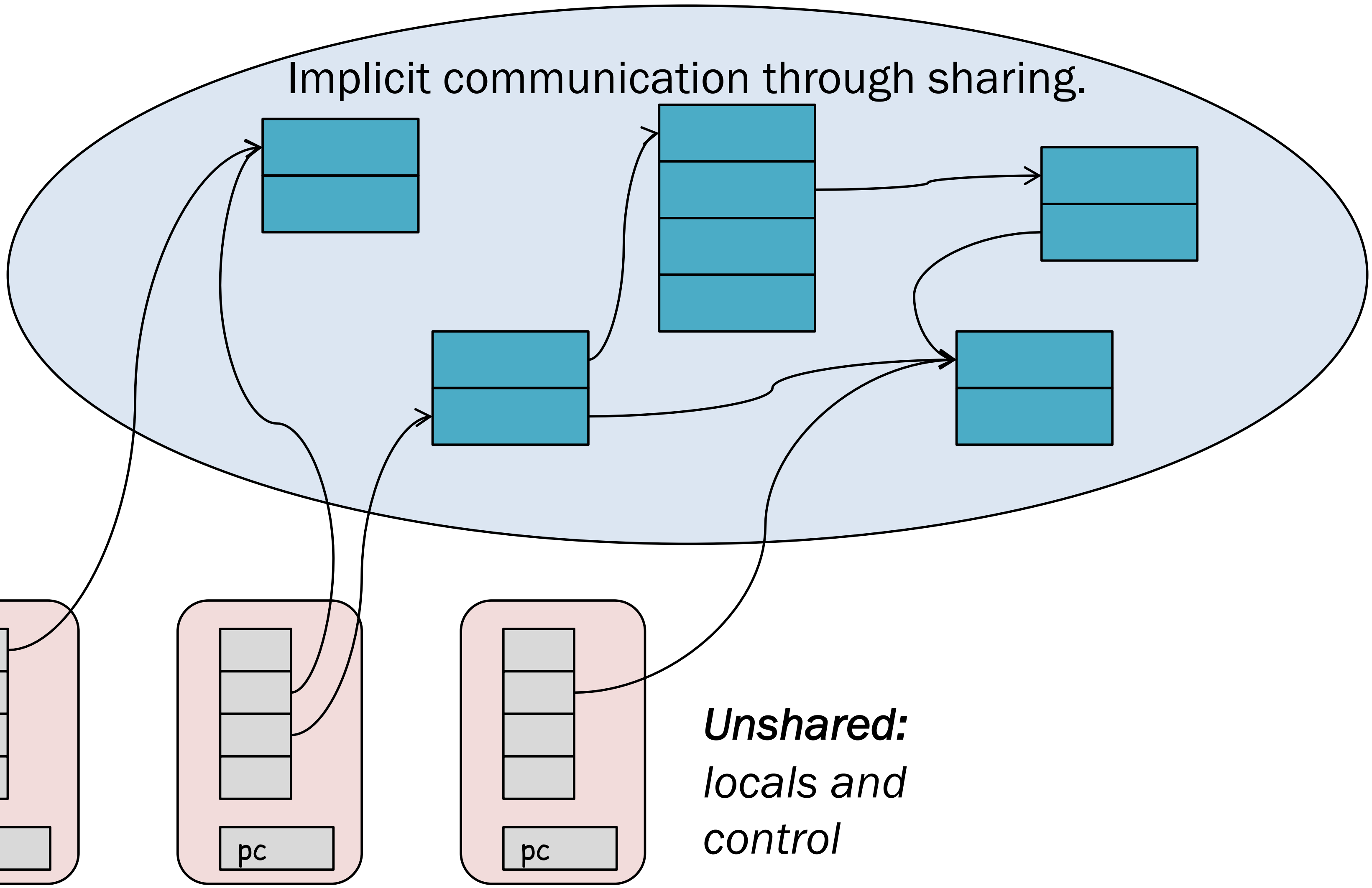
Other models:

Shared-memory multithreading + synchronization

…

Shared-Memory Multithreading

Shared:
heap and globals



Unshared:
locals and control

Concurrency and Race Conditions

```
int bal = 0;
```

Thread 1

```
t1 = bal
bal = t1 + 10
```

Thread 2

```
t2 = bal
bal = t2 - 10
```

Thread 1	Thread 2
`t1 = bal`	
`bal = t1 + 10`	
	`t2 = bal`
	`bal = t2 - 10`

bal == 0

Concurrency and Race Conditions

```
int bal = 0;
```

Thread 1

```
t1 = bal
bal = t1 + 10
```

Thread 2

```
t2 = bal
bal = t2 - 10
```

Thread 1	Thread 2
`t1 = bal`	
	`t2 = bal`
`bal = t1 + 10`	
	`bal = t2 - 10`

bal == -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
}
```

Thread 1

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

Thread 2

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