Parallelism
(and Concurrency)

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

Eliminate 1 big assumption:
Evaluation happens as a sequence of ordered steps.

Parallelism
Use more resources to complete work faster.

Concurrency
Coordinate access to shared resources.

Both can be expressed using a variety of primitives.
Parallelism via Manticore

- Extends SML with language features for parallelism/concurrency.
- Mix research vehicle / established models.
- Parallelism patterns:
  - data parallelism:
    - parallel arrays
    - parallel tuples
  - task parallelism:
    - parallel bindings
    - parallel case expressions
- Unifying model:
  - futures / tasks
- Mechanism:
  - work-stealing

Data parallelism

- many argument data of same type
- parallelize application of same operation to all data
- many result data of same type
- no ordering/interdependence

Parallel arrays: 'a parray

| [ | e1, e2, ..., en | ] | literal parray |
| [ | elo to ehi by estep | ] | integer ranges |
| [ | e | x in elems | ] | parallel mapping comprehensions |
| [ | e | x in elems where pred | ] | parallel filtering comprehensions |

Parallel array comprehensions

| [ | e1 | x in e2 | ] |

Evaluation rule:
1. Under the current environment, E, evaluate e2 to a parray v2.
2. For each element vi in v2, with no constraint on relative timing order:
   1. Create new environment Ei = x→vi, E.
   2. Under environment Ei, evaluate e1 to a value vi'
3. The result is [ | v1', v2', ..., vn' | ]
**Parallel map / filter**

\[
\text{fun mapP } f \text{ xs } = \ [ \mid f \ x \mid x \ \text{in} \ xs \mid ]
\]

: (a -> 'b) -> a array -> 'b array

\[
\text{fun filterP } p \text{ xs } = \ [ \mid x \mid x \ \text{in} \ xs \ \text{where} \ p \ x \mid ]
\]

: (a -> bool) -> a array -> 'a array

---

**Task parallelism**

Parallelize application of different operations within larger computation

Some ordering/interdependence controlled explicitly

---

**Parallel reduce**

\[
\text{fun reduceP } f \text{ init } xs = \ldots
\]

: ((a * a) -> a) -> a -> a array -> a

**Parallel bindings**

\[
\text{fun qsortP } (a: \text{int array}) : \text{int array} =
\begin{align*}
\text{if } & \text{lengthP } a \leq 1 & \text{then } & a \\
\text{else } & \text{let } & \text{val } & \text{pivot } = a ! 0 (* \text{array indexing} *) \\
& & \text{val } & \text{sorted_lt } = \text{qsortP } (\text{filterP } (f n x \Rightarrow x < \text{pivot}) \ a) \\
& & \text{val } & \text{sorted_eq } = \text{filterP } (f n x \Rightarrow x = \text{pivot}) \ a \\
& & \text{val } & \text{sorted_gt } = \text{qsortP } (\text{filterP } (f n x \Rightarrow x > \text{pivot}) \ a) \\
& & \text{in } & \text{concatP } (\text{sorted_lt}, \text{concatP } (\text{sorted_eq}, \text{sorted_gt})) \\
& & \text{end}
\end{align*}
\]

Start evaluating in parallel but don’t wait until needed.

Wait until results are ready before using them.
Manticore

**Parallel cases**

```ml
datatype 'a bintree = Empty
  | Node of 'a * 'a bintree * 'a bintree

fun find_any t e =
  case t of
    Empty => NONE
  | Node (elem, left, right) =>
    if e = elem then SOME t
    else
      pcase find_any left e & find_any right e of
    SOME tree & ? => SOME tree
    | ? & SOME tree => SOME tree
    | NONE & NONE => NONE
```

Evaluate these in parallel.

If one finishes with SOME, return it without waiting for the other.

If both finish with NONE, return NONE.

---

**Futures: unifying model for Manticore parallel features**

```ml
signature FUTURE =
sig
  type 'a future

  val future : (unit -> 'a) -> 'a future

  val touch : 'a future -> 'a

  (* More advanced features. *)
  datatype 'a result = VAL of 'a | EXN of exn

  val poll : 'a future -> 'a result option

  val cancel : 'a future -> unit
end
```

future = promise speculatively forced in parallel

---

**Futures: timeline visualization 1**

```ml
let
  val f = future (fn () => e)

in
  work

  (touch f)

end
```

---

**Futures: timeline visualization 2**

```ml
let
  val f = future (fn () => e)

in
  work

  (touch f)

end
```
**pval as future sugar**

```
let pval x = e
  in ... x ... end
```

```
let val x = future (fn () => e)
  in ... (touch x) ... end
```

*a bit more: implicitly cancel an untouched future once it becomes clear it won’t be touched.

---

**Parray ops as futures: rough idea 1**

Suppose we represent parrays as lists* of elements:

```
[ | f x | x in xs | ]
```

```
map touch
  (map (fn x =>
        future (fn () => f x))
      xs)
```

*actual implementation uses a more sophisticated data structure

---

**Parray ops as futures: rough idea 2**

Suppose we represent parrays as lists* of element futures:

```
[ | f x | x in xs | ]
```

```
map (fn x => future
      (fn () => f (touch x)))
      xs
```

*actual implementation uses a more sophisticated data structure

---

**Odds and ends**

- **pcase**: not just future sugar
  - Choice is a distinct primitive* not offered by futures alone.
- Where do execution resources from futures come from? How are they managed?
- Tasks vs futures:
  - Analogy: function calls vs. val bindings.
- Forward to concurrency and events...

*at least when implemented well.