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

no ordering/interdependence

many result data of same type
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 $e_2$ to a parray $v_2$.
2. For each element $v_i$ in $v_2$, with no constraint on relative timing order:
   1. Create new environment $E_i = x \mapsto v_i, E$.
   2. Under environment $E_i$, evaluate $e_1$ to a value $v_i'$
3. The result is [ | $v_1'$, $v_2'$, ..., $v_n'$ | ]
Parallel map / filter

fun mapP f xs =

    [| f x | x in xs |]

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

fun filterP p xs =

    [| x | x in xs where p x |]

: ('a -> bool) -> 'a parray -> 'a parray
Parallel reduce

`sibling of foldl/foldr` combiner function, $f$, must be **associative**

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

\[
: (('a \times 'a) \to 'a) \to 'a \to 'a \text{ parray } \to 'a
\]
Task parallelism

parallelize application of different operations within larger computation

some ordering/interdependence controlled explicitly
fun qsortP (a: int parray) : int parray =
  if lengthP a <= 1
  then a
  else
    let
      val pivot = a ! 0 (* parray indexing *)
pval sorted_lt =
    qsortP (filterP (fn x => x < pivot) a)
pval sorted_eq =
    filterP (fn x => x = pivot) a
pval sorted_gt =
    qsortP (filterP (fn x => x > pivot) a)
in
  concatP (sorted_lt, concatP (sorted_eq, sorted_gt))
end

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

Wait until results are ready before using them.
Parallel cases

```ocaml
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

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

    (* Produce a future for a thunk. 
       Like Promise.delay. *)
    val future : (unit -> 'a) -> 'a future

    (* Wait for the future to complete and return the result. 
       Like Promise.force. *)
    val touch : 'a future -> 'a

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

    (* Check if the future is complete and get result if so. *)
    val poll : 'a future -> 'a result option

    (* Stop work on a future that won't be needed. *)
    val cancel : 'a future -> unit
  end
```

future = promise speculatively forced in parallel
Futures: timeline visualization 1

let
  val f = future (fn () => e)
in
work

... (touch f)...
end
let
  val f = future (fn () => e)
in
work
...
(touch f)
...
end
pval as future sugar

```plaintext
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

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

Suppose we represent parrays as lists* of element futures:

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

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

Key semantic difference 1 vs 2?

*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.*