Parallelism

(and Concurrency)

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

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.
- Coordinate access to shared resources.

Concurrency

- 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

Parallel Arrays: 'a parray

\[ [\mid e_1, e_2, \ldots, e_n \mid] \]

\[ [\mid e \mid x \in \text{elems} \mid] \]

\[ [\mid e \mid x \in \text{elems where pred} \mid] \]

parallel mapping comprehensions

parallel filtering comprehensions

parallel array comprehensions

\[ [\mid e_1 \mid x \in \text{e}_2 \mid] \]

Evaluation rule:
1. Under the current environment, \( E \), evaluate \( \text{e}_2 \) to a parray \( \text{v}_2 \).
2. For each element \( \text{v}_i \) in \( \text{v}_2 \), with no constraint on relative timing order:
   1. Create new environment \( E_i = x \rightarrow \text{v}_i, E \).
   2. Under environment \( E_i \), evaluate \( e_1 \) to a value \( \text{v}_i' \).
3. The result is \[ [\mid \text{v}_1', \text{v}_2', \ldots, \text{v}_n' \mid] \]

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 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

Task Parallelism

parallelize application of different operations within larger computation

some ordering/interdependence controlled explicitly

Parallel Reduce

fun reduceP f init xs = ...

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

sibling of fold

f must be associative

Parallel Bindings

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.
**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

**signature FUTURE =**

```ml
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

  (* 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

```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**

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

```ml
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 | ]
```

```ml
touch x
```

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

*not the actual implementation

---

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

Suppose we represent parrays as lists* of element futures:

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

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

*not the actual implementation

---

**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:
  - function calls vs. val bindings.
- Forward to concurrency and events...

*not the actual implementation

---

*at least when implemented well.