Teaching Parallelism to Freshmen

Robert Harper

December, 2011

Premises

Parallelism abounds!

- Multicores.
- Distribution.
- Graphics processors.

Parallelism is about efficiency, not correctness.

- Not about concurrency!
- Determinacy = sequential semantics, parallel cost.

Teaching Parallelism

Functional programming

- Computation by transformation.
- Persistent, not ephemeral, data structures.
- Manipulate aggregates as a whole: death to iterators!

Cost semantics

- Work = sequential complexity.
- Span = idealized parallel complexity (critical path length).
- Brent's Principle: bound performance based on cost.

Machine Models

Traditionally, algorithms research has focused on machine models.

- Sequential RAM.
- Parallel RAM with various capabilities.
- Relentlessly imperative. No abstraction.

Cost is derived from (fictional) compilation of HLL onto RAM.

- Reason about the compiled code (with hand-waving).
- Bakes in number of processors, assumptions about interconnect.

procedure QUICKSORT(**S**):

  if S contains at most one element **then return S**

  else

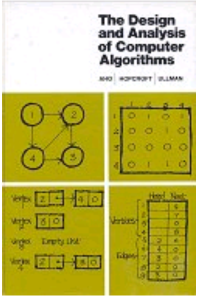
    begin

      choose an element **a** randomly from **S**;

      let $\mathbf{S_1}$, $\mathbf{S_2}$ and $\mathbf{S_3}$ be the sequences of elements in **S** less than, equal to, and greater than **a**, respectively;

      return (QUICKSORT($\mathbf{S_1}$) followed by $\mathbf{S_2}$ followed by QUICKSORT($\mathbf{S_3}$))

end



Language Models

Ironically, the classic AHU Quicksort is specified cleanly!

- Naturally parallel.
- No low-level details.

But conventional (especially, commercial) languages are relentlessly low-level.

- Machine-inspired imperative model.
- OOPL's don't help, they make the problem worse!

Language Models

What is a language-based model of parallel computation?

- A static semantics that specifies the well-formed programs.
- A dynamic semantics that specifies both the execution and the cost of a program.
- A provable implementation that realizes the abstract cost on a concrete machine model.

Crucially,

- The execution semantics is not affected by parallelism.
- The cost semantics specifies both sequential and parallel complexity.
- The provable implementation takes account of scheduling and interconnect costs.

Parallel Functional Programming

Evaluation semantics: $e \Downarrow v$.

$$\frac{}{\lambda x{:}\tau.e \Downarrow \lambda x{:}\tau.e}$$

$$\frac{e_1 \Downarrow \lambda x{:}\tau.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\, e_2 \Downarrow v}$$

Evaluation, not execution: no effects, no interference!

Parallel Functional Programming

Cost semantics: $e \Downarrow^{d}_{w} v$.

- Work, w, is total number of steps (sequential complexity).
- Depth (aka span), d, is critical path length (idealized parallel complexity).

$$\frac{}{\lambda x{:}\tau.e \Downarrow^{1}_{1} \lambda x{:}\tau.e}$$

$$\frac{e_1 \Downarrow^{d_1}_{w_1} \lambda x{:}\tau.e \quad e_2 \Downarrow^{d_2}_{w_2} v_2 \quad [v_2/x]e \Downarrow^{d_3}_{w_3} v}{e_1\ e_2 \Downarrow^{\max(d_1,d_2)+d_3+1}_{w_1+w_2+w_3+1} v}$$

Specifies parallelism by specifying cost model!

Parallel Asymptotics

The cost model allows us to assign complexity to programs.

- $T_1(n) =$ sequential execution time for input of size n.
- $T_\infty(n) =$ idealized parallel execution time for input of size n.

Using this we can assess the parallelizability of an algorithm.

Quicksort of a List

Sequential partition, parallel recursive calls:

```
fun qs [] = []
  | qs (xs as a::_) =
      let val ls = filter (fn x => x<a) xs
          val es = filter (fn x => x=a) xs
          val gs = filter (fn x => x>a) xs
      in (qs ls) @ es @ (qs gs) end
```

Complexity (expected, over all inputs of size n):

$$T_1(n) = O(n \log n)$$
$$T_\infty(n) = O(n)$$

Not very parallelizable!

Quicksort on a Tree

```
datatype 'a seq = Empty
  | Leaf of 'a
  | Node of 'a seq * 'a seq

fun app Empty b = b
  | app a Empty = a
  | app a b = Node (a, b)

fun fil p Empty = Empty
  | fil p (Leaf x) = if p x then Leaf x else Empty
  | fil p (Node (a, b)) = Node (fil p a, fil p b)
```

Quicksort on a Tree

```
fun qs Empty = Empty
  | qs xs =
    let val a = head xs
        val ls = fil (fn x => x<a) xs
        val es = fil (fn x => x=a) xs
        val gs = fil (fn x => x>a) xs
    in  app (qs ls) (app es (qs gs)) end
```

Complexity:

- $T_1(n) = O(n \log n)$
- $T_\infty(n) = O(\log^2 n)$.

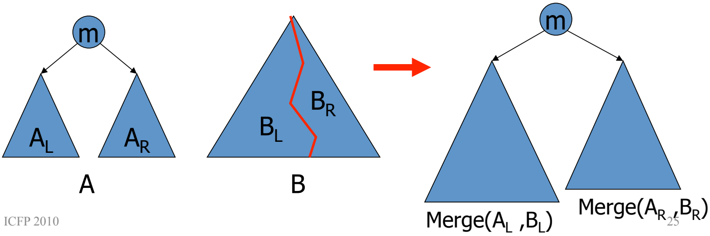
Parallelizable!

Parallel Merge

Span = $O(\log^2 n)$
Work = $O(n)$

```
Merge(A,B) =
  let
    Node(AL, m, AR) = A
    (BL ,BR) = split(B, m)
  in
    Node(Merge(AL,BL), m, Merge(AR,BR))
```

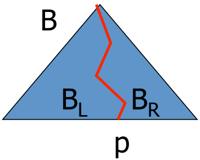
Merge in parallel



Parallel Merge

```
datatype 'a seq = Empty
                | Node of 'a * 'a seq * 'a seq

fun split (p, Empty) = (Empty, Empty)
  | split (p, node(v, L, R)) =
    if p < v then
      let val (L1 ,R1) = split(p ,L)
      in (L1,node(v, R1, R)) end
    else
      let val (L1,R1) = split(p ,R)
      in (node (v, L, L1), R1) end;
```



Provable Implementation

Theorem (Brent; Blelloch and Greiner)

If $e \Downarrow_w^d v$*, then* v *can be calculated on a CREW PRAM with* p *processors in time* $O(\max(w/p, d \log p))$*.*

The $\lg p$ factor accounts for the interconnect.

The proof is essentially a scheduler for the parallel tasks, using Brent's Principle.

- Do work in chunks of w/p insofar as possible.
- Critical path length imposes a lower bound of d steps.

Parallelizability

The parallelizability ratio is (by definition) $T_1(n)/T_\infty(n)$.

- Parallelizable if ratio is larger than p.
- Not parallelizable otherwise.
- Can compare ratios for different algorithms.

Provides a metric for assessing the potential to exploit parallelism in a given program.

Collections

Evaluation of collections in parallel:

$$\frac{e \Downarrow [v_1, \ldots, v_n] \quad [v_1/x]e' \Downarrow v'_1 \quad \ldots \quad [v_n/x]e' \Downarrow v'_n}{\{\, e' : x \in e \,\} \Downarrow [v'_1, \ldots, v'_n]}$$

Cost semantics:

$$\frac{e \Downarrow^{d}_{w} [v_1, \ldots, v_n] \quad [v_1/x]e' \Downarrow^{d_1}_{w_1} v'_1 \quad \ldots \quad [v_n/x]e' \Downarrow^{d_n}_{w_n} v'_n}{\{\, e' : x \in e \,\} \Downarrow^{d+\max(d_1,\ldots,d_n)+1}_{w+w_1+\cdots+w_n+1} [v'_1, \ldots, v'_n]}$$

Fork n threads, one for each element, store results in pre-allocated array.

Matrix Multiplication

```
fun mm A B =
  let
    fun vv b a =
      red (+) (fn (i,x) => sub(b,i)*x) 0.0 a
    fun mv B a =
      tab (fn i => vv (sub (B,i)) a)) (len B)
  in
      tab (fn i => mv B (sub (A, i)) (len A)
  end
```

New Curriculum at CMU

15-150: Functional Programming (Harper, Licata).

- Computing by transformation.
- Persistent, as well as ephemeral, data structures.
- Verification of correctness.
- Parallel thinking: cost semantics, aggregates.
- Modularity and abstraction.
- Example: Barnes-Hut.

See: http://www.cs.cmu.edu/~15150

New Curriculum at CMU

15-210: Parallel Data Structures and Algorithms (Blelloch).

- Complete re-boot of classical DS+A course: no objects, no pointers, no machine models.
- Functional programming over persistent data structures.
- Abstract types: separating abstraction from implementation.
- Asymptotics: work and depth.
- Example: shotgun method for genome sequencing.

See: http://www.cs.cmu.edu/~15210

New Curriculum at CMU

15-122: Imperative Programming (Pfenning).

- C0, a safe C-like language (aka Pascal with curly braces).
- Emphasize verification, run-time checking.
- Classic pointer mentality, including null's.
- No parallelism.

Questions?

Thanks for your attention!