Cost Models based on the λ-Calculus or The Church Calculus the Other Turing Machine

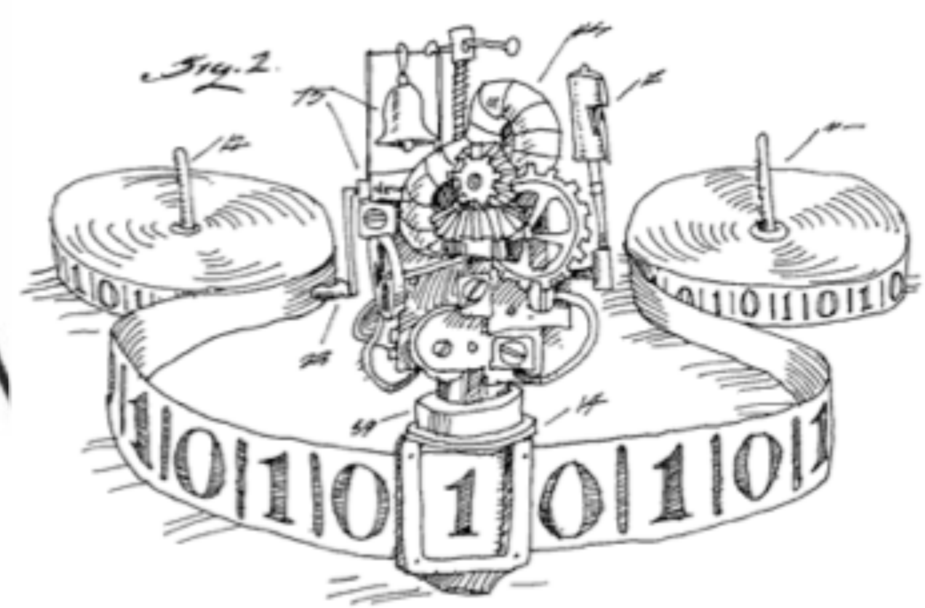
Guy Blelloch

Carnegie Mellon University

Church/Turing Hypothesis

$$(\lambda x.\, e_1)\, e_2 \ \Rightarrow_{\beta}\ e_1\, [e_2 / x\,]$$

$$=$$



Machine Models and Simulation

Handbook of Theoretical Computer Science

Chapter 1: Machine Models and Simulations

[Peter van Emde Boas]

Machine Models [2nd paragraph]

“If one wants to reason about complexity measures such as **time and space** consumed by an **algorithm**, then one must specify precisely what notions of time and space are meant.

The **conventional notions** of time and space complexity within theoretical computer science are based on the implementation of algorithms on abstract ma[illegible]alled **machi**[illegible]**els**.”

language-based cost models

Simulation [3rd paragraph]

"Even if we base complexity theory on abstract instead of concrete machines, the arbitrariness of the choice of model remains. It is at this point that the notion of simulation enters. If we **present mutual simulations** between two models and give estimates for the **time and space overheads** incurred by performing these simulations…"

Machine Models

Goes on for over 50 pages on machine models

Turing Machines

- 1 tape, 2 tape, m tapes
- 2 stacks
- 2 counter, m counters,
- multihead tapes,
- 2 dimensional tapes
- various state transitions

Machine Models

Random Access Machines

- SRAM (succ, pred)
- RAM (add, sub)
- MRAM (add, sub, mult)
- LRAM (log length words)
- RAM-L (cost of instruction is word length)

Pointer Machines

- SMM, KUM, pure, impure

Several others

Some Simulation Results (Time)

- SRAM(time n) < TM(time n^2 log n)
- RAM(time n) < TM(time n^3)
- RAM-L(time n) < TM(time n^2)
- LRAM(time n) < TM(time n^2 log n)
- MRAM(time n) < TM(time Exp)

- TM(time n) < SRAM(time n)
- TM(time n) < RAM(time n/log n)

Parallel Machine Models

- Circuit models
- PSPACE
- TM with alternation
- Vector models
- PRAM
  - EREW, CREW, CRCW (priority, arbitrary, …)
- SIMDAG
- k-PRAM, MIND-RAM, PTM

The two parts

Part 1: The model

- Well defined semantics
- Simple
- Close to programming paradigm

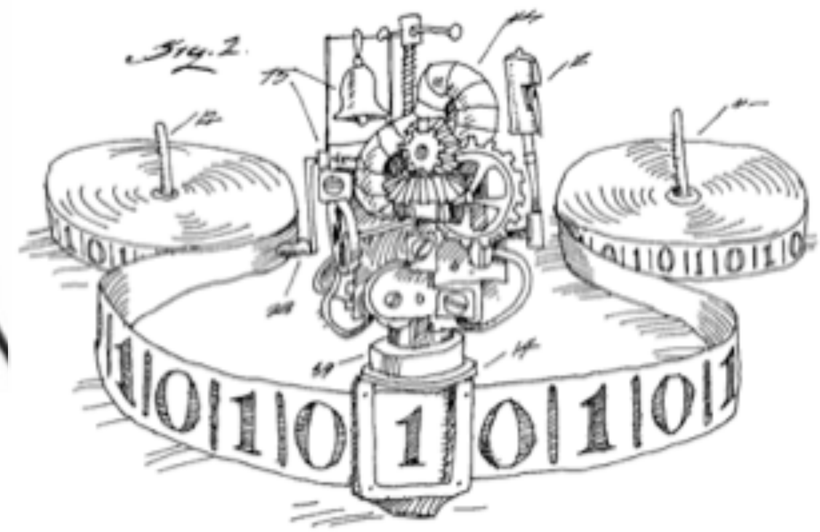
Part 2: Simulatation

- Mapping of costs
- Good bounds when simulated on realistic machines

Church/Turing

$$(\lambda x.\, e_1)\, e_2 \;\Rightarrow_\beta\; e_1\,[e_2 / x\,]$$

$$=$$



For Costs?

Language-Based Cost Models

A cost model based on a "cost semantics" instead of a machine.

Why use the λ-calculus? historically the first model, a very clean model, well understood.

What costs? Number of reduction steps is the simplest cost, but as we will see, not sufficient (e.g. space, parallelism).

Language-Based Cost Models

Advantages over machine models:

- naturally parallel (parallel machine models are messy)
- more elegant
- model is closer to code and algorithms
- closer in terms of simulation costs to "practical" machine models such as the RAM.

Disadvantages:

- 50 years of history

Work in this area

- SECD machine [Landin 1964]
- CBN, CBV and the λ-Calculus [Plotkin 1975]
- Cost Semantics [Sands, Roe, ….]
- Call-by-value λ-Calculus [BG 1995, FPGA]
- CBV λ-Calculus with Arrays [BG 1996,ICFP]
- Call-by-need/speculation [BG 1996, POPL]
- Various recent work [Martini, Dal Lago, Accattoli, SGM,…
- CVB cache model [BH 2012, POPL]

Work in this area

- SECD machine [Landin 1964]
- CBN, CBV and the λ-Calculus [Plotkin 1975]
- Cost Semantics [Sands, Roe, ….]
- Call-by-value λ-Calculus [BG 1995, FPGA]
- CBV λ-Calculus with Arrays [BG 1996,ICFP]
- Call-by-need/speculation [BG 1996, POPL]
- Various recent work [Martini, Dal Lago, Accattoli, SGM,…
- CVB cache model [BH 2012, POPL]

Call-by-value λ-calculus

$$e = x \mid (e_1 \; e_2) \mid \lambda x.\, e$$

Call-by-value λ-calculus

$e \Downarrow v$ relation

$$\lambda x.\, e \Downarrow \lambda x.\, e \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\, e \qquad e_2 \Downarrow v \qquad e[v/x] \Downarrow v'}{(e_1\ e_2) \Downarrow v'} \ \text{(APP)}$$

The λ-calculus is Parallel

$$\frac{e_1 \Downarrow \lambda x.\, e \qquad e_2 \Downarrow v \qquad e[v/x] \Downarrow v'}{e_1\; e_2 \Downarrow v'}\ \text{(APP)}$$

It is “safe” to evaluate e_1 and e_2 in parallel

But what is the cost model?

How does it compare to other parallel models?

Part 1: Cost Model

$$e \Downarrow v; w, d$$

Reads: expression *e* evaluates to *v* with work *w* and span *d.*

- **Work** (W): sequential work
- **Span** (D): parallel depth

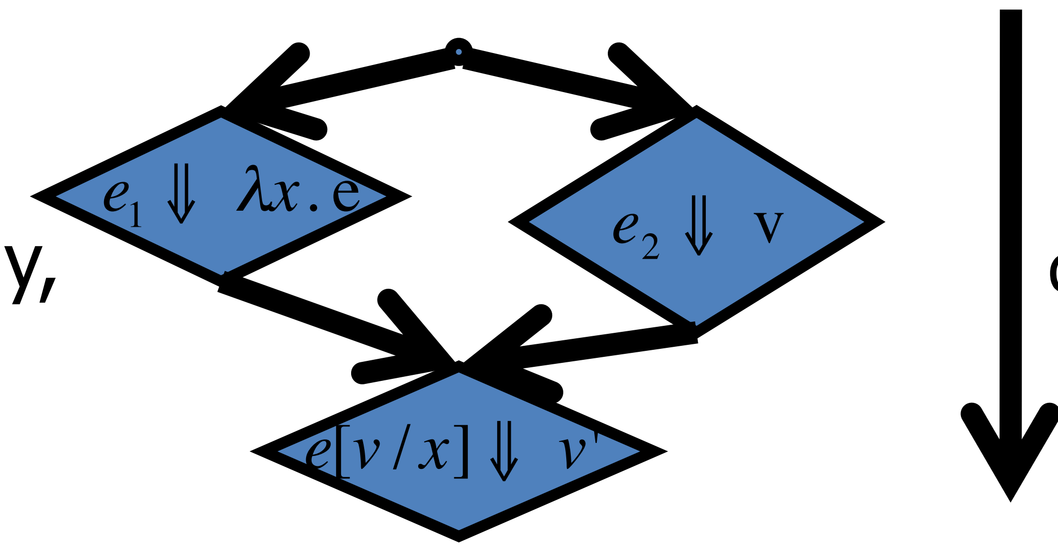
Span captures dependence depth

The Parallel λ-calculus: cost model

$$\lambda x.e \Downarrow \lambda x.e; 1, 1 \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\,e; w_1, d_1 \quad e_2 \Downarrow v; w_2, d_2 \quad e[v/x] \Downarrow v'; w_3, d_3}{e_1\, e_2 \Downarrow v'; 1 + w_1 + w_2 + w_3, 1 + \max(d_1, d_2) + d_3} \qquad \text{(APP)}$$

Work adds
Span adds sequentially,
and max in parallel



The Parallel λ-calculus: cost model

$$\lambda x.\mathrm{e} \Downarrow \lambda x.\mathrm{e}; 1, 1 \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\, e;\, w_1, d_1 \quad e_2 \Downarrow v;\, w_2, d_2 \quad e[v/x] \Downarrow v';\, w_3, d_3}{e_1\, e_2 \Downarrow v';\, 1 + w_1 + w_2 + w_3,\, 1 + \max(d_1, d_2) + d_3} \qquad \text{(APP)}$$

let, letrec, datatypes, tuples, case-statement can all be implemented with constant overhead

Integers and integer operations (+, <, …) can be added as primitives or implemented with O(log n) cost.

Defining basic types and constructs

Recursive Data types

pair $\equiv \lambda\, x\, y.(\lambda f.f\, x\, y))$

first $\equiv \lambda p.p\ (\lambda\, x\, y.\, x)$ **second** $\equiv \lambda p.p\ (\lambda\, x\, y.\, y)$

Local bindings

let val x = e_1 **in** e **end** $\equiv$ $(\lambda\, x\, .\, e)\ e_1$

Conditionals

true $\equiv \lambda\, x\, y.\, x$ **false** $\equiv \lambda\, x\, y.\, y$

if e_1 **then** e_2 **else** e_3 $\equiv$ $((\lambda p\, .\, (\lambda\, x\, y\, .\, p\, x\, y))\ e_1)\ e_2\ \ e_3$

Recursion

Y-combinator (CBV version)

Integers (logarithmic overhead)

List of bits (true/false values)

Church numerals do not work in CBV

Part 2: The Simulation

Simulate on a RAM (sequential) or PRAM (parallel).

- What about cost of substitution, or variable lookup?
- What about finding a redux?

Can be done efficiently

- implement with sharing via a store or environment. If using an environment variable lookup is “cheap”

Simulation (sequential)

CEK machine : a state transition system

$$(C, E, K) \Rightarrow (C', E', K')$$

“control” C := $e_1 e_2 \mid \lambda x.\, e \mid x$

“environment” $E \coloneqq x \ \rightarrow v$

“continuation” K := done | arg(e,E,K) | fun(e,E,K)

Simulation (sequential)

CEK machine : a state transition system

$$(C, E, K) \Rightarrow (C', E', K')$$

$$(e_1\, e_2, E, K) \Rightarrow (e_1\,, E, \mathrm{arg}(e_2, E, K))$$
$$(x, E, K) \Rightarrow (E(x), E, K)$$
$$(v, E, \mathrm{arg}(e, E', K)) \Rightarrow (e\,, E', fun(v, E, K))$$
$$(v, E, fun(\lambda x.\, e, E', K)) \Rightarrow (e\,, E' + (x \rightarrow v), K)$$

Simulation (parallel)

P-CEK machine : another state transition system

$\langle (C_1, E_1, K_1), (C_2, E_2, K_2), \ldots \rangle \Rightarrow \langle (C'_1, E'_1, K'_1), (C'_2, E'_2, K'_2), \ldots \rangle$

Each step makes multiple operations in parallel:

$(e_1\ e_2, E, K) \Rightarrow \langle (e_1\ , E, \mathrm{fun}(l, K)), (e_2\ , E, \mathrm{arg}(l, K)) \rangle$ new l

Need to ”synchronize” on l

Part 2: The Simulation Bounds

Theorem [FPCA95]:If $e \Downarrow v; w, d$ then v can be calculated from e on a CREW PRAM with p processors in $O\left(\frac{w}{p} + d \log p\right)$ time.

Can't really do better than: $\max\left(\frac{w}{p}, d\right)$

If w/p > d log p then "work dominates"

We refer to w/p as the parallelism.

The Parallel λ-calculus (including constants)

$$c \Downarrow c; 1, 1 \quad \text{(CONST)}$$

$$\frac{e_1 \Downarrow c; w_1, d_1 \quad e_2 \Downarrow v; w_2, d_2 \quad \delta(c, v) \Downarrow v'}{e_1\ e_2 \Downarrow v'; 1 + w_1 + w_2, 1 + \max(d_1, d_2)} \quad \text{(APPC)}$$

$$c_n = 0, \cdots, n, +, +_0, \cdots, +_n, <, <_0, \cdots, <_n, \times, \times_0, \cdots, \times_n, \cdots \quad \text{(constants)}$$

Quicksort in the λ-Calculus

```
fun qsort S =
  if (size(S) <= 1) then S
  else
    let val a = randelt S
        val S1 = filter (fn x => x < a) S
        val S2 = filter (fn x => x = a) S
        val S3 = filter (fn x => x > a) S
    in
      append (qsort S1) (append S2 (qsort S3))
    end
```

Qsort on Lists

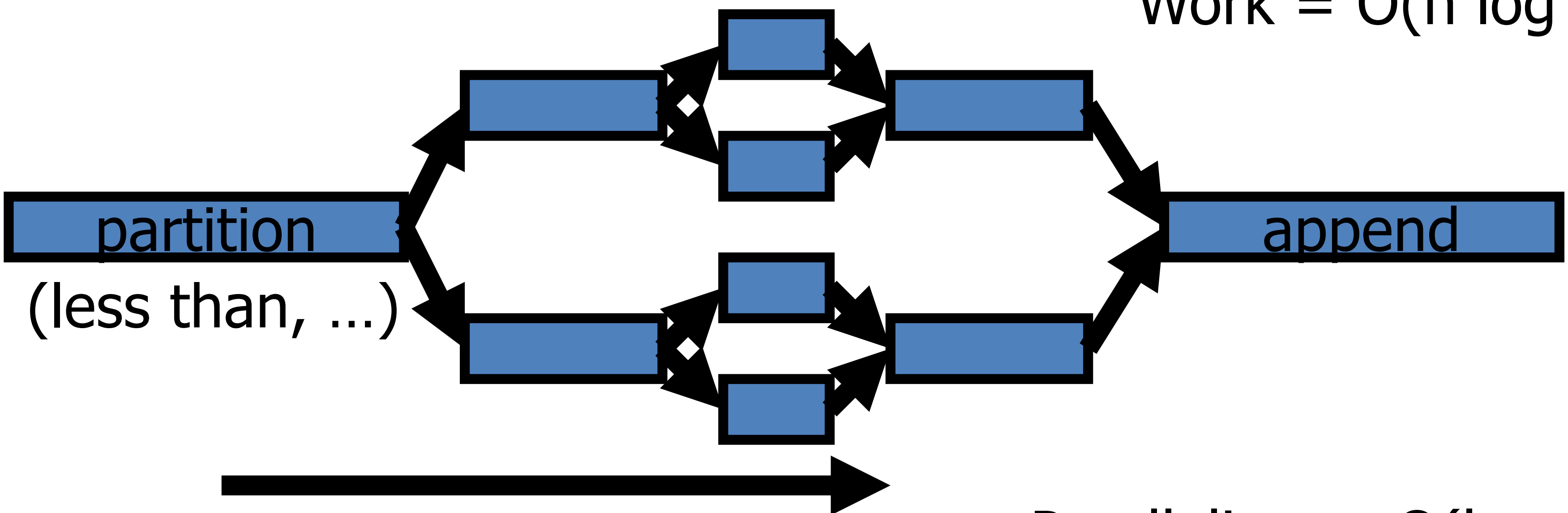
```
fun qsort [] = []
  | qsort S =
    let val a::_ = S
        val S1 = filter (fn x => x < a) S
        val S2 = filter (fn x => x = a) S
        val S3 = filter (fn x => x > a) S
    in
      append (qsort S1) (append S2 (qsort S3))
    end
```

Qsort Complexity

Sequential Partition
Parallel calls

All bounds expected case over all inputs of size n

Work = O(n log n)



Span = O(n)

Parallelism = O(log n)

Not a very good parallel algorithm

Tree Quicksort

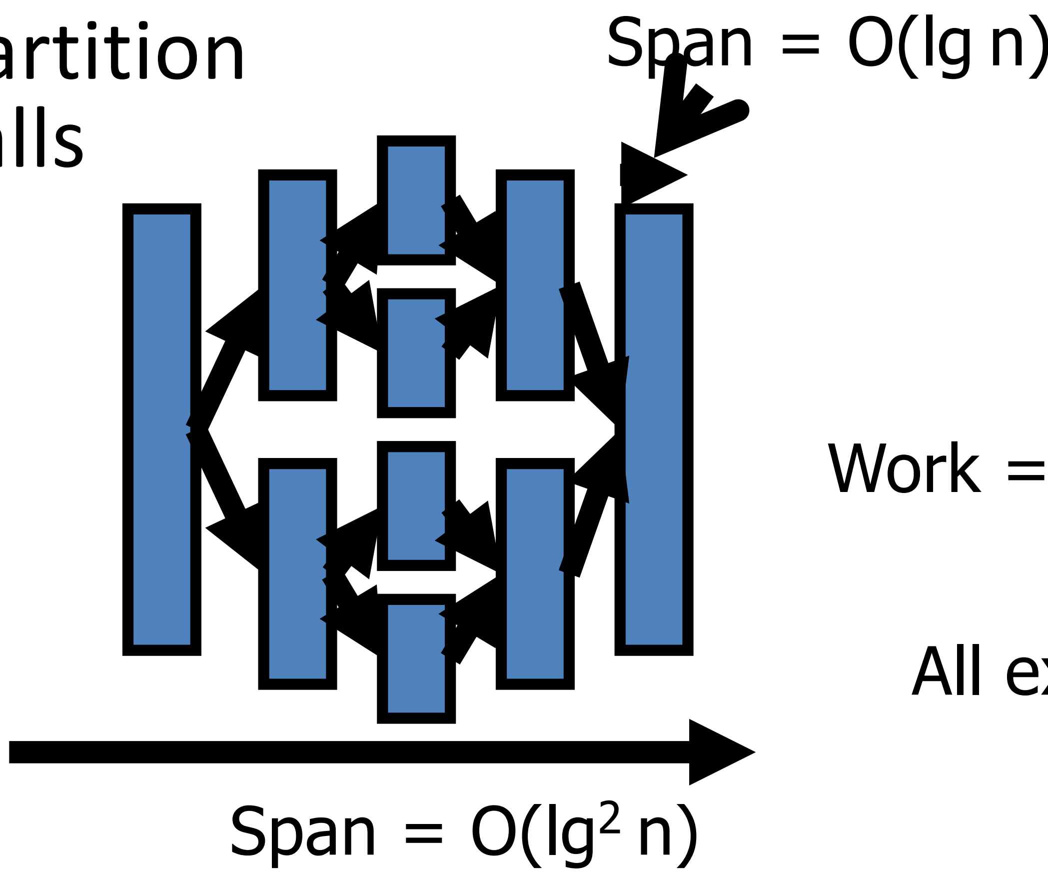
```
datatype 'a seq = Empty
                | Leaf of 'a
                | Node of 'a seq * 'a seq

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

fun filter f Empty = Empty
  | filter f (Leaf x) =
     if (f x) the Leaf x else Empty
  | filter f Node(l,r) =
     append (filter f l) (filter f r)
```

Tree Quicksort

```
fun qsort Empty = Empty
  | qsort S =
    let val a = first S
        val S1 = filter (fn x => x < a) S
        val S2 = filter (fn x => x = a) S
        val S3 = filter (fn x => x > a) S
    in
      append (qsort S1) (append S2 (qsort S3))
    end
```

Qsort Complexity

Parallel partition
Parallel calls



Work = O(n log n)

All expected case

A good parallel algorithm

Parallelism = O(n/log n)

Remember: Cost Composition

	Work	Span
Sequential	Add	Add
Parallel	Add	Max

Recurrences For Divide and Conquer:

$$W(n) = 2W\left(\frac{n}{2}\right) + W_{split}(n) + W_{join}(n)$$

$$S(n) = S\left(\frac{n}{2}\right) + S_{split}(n) + S_{join}(n)$$

Cost Composition: Example

Mergesort:

Merge: $W(n) = O(n)$, $S(n) = O(\log n)$

Mergesort:

$$W(n) = 2\,W(n/2) + O(n)$$
$$= O(n \log n)$$
$$S(n) = S(n/2) + O(\log n)$$
$$= O(\log^2 n)$$

Adding Functional Arrays: NESL

tabulate f e

$$\frac{f \Downarrow \lambda x. e'; w, d \qquad e \Downarrow n; w', d'' \qquad e'[i\ for\ x] \Downarrow v'_i; w_i, d_i \quad i \in \{1..n\}}{tabulate\ f\ e \Downarrow [v'_1 \ldots v'_n]; w + w' + \sum w_i, d + d' + \max(d_i)}$$

Other Primitives:

```
<- : 'a seq * (int,'a) seq -> 'a seq
```

- `[q,p,x,y,s] <- [(0, o),(3,s),(2,l)]`
  `[o,p,l,s,s]`

```
elt, index, length
```

Conclusions

2 parts to a cost mode:

- Model itself
- Simulation results

λ-calculus good for modeling:

- An actual programming modlel
- sequential time (work) – closely matches RAM
- parallel time (nested parallelism) – closely matches PRAM