Cost Models based on the \(\lambda\)-Calculus
or
The Church Calculus
the Other Turing Machine

Guy Blelloch
Carnegie Mellon University
Church/Turing

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

=
Machine Models and Simulation

Handbook of Theoretical Computer Science

Chapter 1: Machine Models and Simulations

[Peter van Emde Boas]
“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 machines, called machine models.”
Machine Models [2\textsuperscript{nd} paragraph]

“If one wants to reason about complexity measures such as \textit{time and space} consumed by an \textbf{algorithm}, then one must specify precisely what notions of time and space are meant. The \textit{conventional notions} of time and space complexity within theoretical computer science are based on the implementation of algorithms on abstract machines, called \textit{machine models}.”
Simulation [3\textsuperscript{rd} 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)

• $\text{SRAM}(\text{time } n) < \text{TM}(\text{time } n^2 \log n)$
• $\text{RAM}(\text{time } n) < \text{TM}(\text{time } n^3)$
• $\text{RAM-L}(\text{time } n) < \text{TM}(\text{time } n^2)$
• $\text{LRAM}(\text{time } n) < \text{TM}(\text{time } n^2 \log n)$
• $\text{MRAM}(\text{time } n) < \text{TM}(\text{time Exp})$

• $\text{TM}(\text{time } n) < \text{SRAM}(\text{time } n)$
• $\text{TM}(\text{time } n) < \text{RAM}(\text{time } n/\log n)$
Some Simulation Results (Space)

- LRAM(space n) < TM(space n log n)
- RAM-L(space n) < TM(space n)
  - space = sum of word sizes, 1 if empty
Complexity Classes

$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE} = \text{NEXPSPACE} \ldots$
Parallel Machine Models

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

\((\lambda x. \, e_1) \, e_2 \Rightarrow_\beta \, e_1 [e_2 / x]\)
Language-Based Cost Models

A cost model based on a “cost semantics” instead of a machine.

Why use the $\lambda$-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
Our work

Call-by-value λ-Calculus [BG 1995, FPGA]
Call-by-need/speculation [BG 1996, POPL]
CBV λ-Calculus with Arrays [BG 1996, ICFP]
CBV space [SBHG 2006, ICFP]
CVB cache model [BH 2012, POPL]

Gibbons, Greiner, Harper, Spoonhower
Other work

• SECD machine [Landin 1964]
• CBN, CBV and the $\lambda$-Calculus [Plotkin 1975]
• Cost Semantics [Sands, Roe, ....]
• The lenient $\lambda$-Calculus [Roe 1991]
• $\lambda$-Calculus and linear speedups [SGM 2002]
• Various recent work [Martini, Dal Lago, Accattoli, ...]
• Various work on “implicit computational complexity” (Leivant, Girard, Cook, ...)

\[ \land \]
Call-by-value $\lambda$-calculus

$$e = x \mid (e_1 \ e_2) \mid \lambda x. \ e$$
Call-by-value $\lambda$-calculus

\[
e \Downarrow v \quad \text{relation}
\]

\[
\lambda x. e \Downarrow \lambda x. e \quad \text{(LAM)}
\]

\[
e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v' \\
\hline
(e_1 \ e_2) \Downarrow v' \quad \text{(APP)}
\]
The λ-calculus is Parallel

\[
\begin{align*}
  e_1 \Downarrow \lambda x. e & \quad e_2 \Downarrow v & \quad e[v/x] \Downarrow v' \\
  \implies e_1 e_2 \Downarrow v'
\end{align*}
\]

(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?
The Parallel $\lambda$-calculus: 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
The Parallel $\lambda$-calculus: cost model

$$\lambda x. e \Downarrow \lambda x. e; \boxed{1, 1}$$  \quad (LAM)

$$\begin{array}{c}
e_1 \Downarrow \lambda x. e; w_1, d_1 \\
e_2 \Downarrow v; w_2, d_2 \\
e[v/x] \Downarrow v'; w_3, d_3 \\
\end{array}$$  \quad (APP)

\[
e_1 e_2 \Downarrow v'; \boxed{1 + w_1 + w_2 + w_3}, 1 + \text{max}(d_1, d_2) + d_3
\]

Work adds
Span adds sequentially, and max in parallel
The Parallel $\lambda$-calculus: cost model

$$\lambda x. e \Downarrow \lambda x. e; 1, 1$$ \hspace{1cm} (LAM)

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

$$e_1 \ e_2 \Downarrow v'; \begin{aligned} & 1 + w_1 + w_2 + w_3, \quad 1 + \max(d_1, d_2) + d_3 \end{aligned}$$

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

Integers and integer operations ($+, <, \ldots$) can be added as primitives or implemented with $O(\log n)$ cost.
Defining basic types and constructs

Recursive Data types

\[
\text{pair} \equiv \lambda x \ y. (\lambda f \ . \ f \ x \ y)
\]
\[
\text{first} \equiv \lambda p \ . \ p \ (\lambda x \ y \ . \ x)
\]
\[
\text{second} \equiv \lambda p \ . \ p \ (\lambda x \ y \ . \ y)
\]

Local bindings

\[
\text{let val } x = e_1 \ \text{in} \ e \ \text{end} \equiv (\lambda x \ . \ e) \ e_1
\]

Conditionals

\[
\text{true} \equiv \lambda x \ y \ . \ x \quad \text{false} \equiv \lambda x \ y \ . \ y
\]
\[
\text{if } e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 \equiv ((\lambda p \ . \ (\lambda x \ y \ . \ p \ x \ y)) \ e_1) \ e_2 \ e_3
\]

Recursion

Y-combinator

Integers (logarithmic overhead)

List of bits (true/false values)

Church numerals do not work
Other costs

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

Not a problem
  - implement with sharing via a store or environment. If using an environment variable lookup is “cheap”
Simulation

- P-CEK machine

\[ \langle (C_1, E_1, K_1), (C_2, E_2, K_2), \ldots \rangle \]

\[ K = \text{nil} | (\text{arg } l :: K) | (\text{fun } l :: K) \]

\[ (e_1 e_2, E, K) \Rightarrow \langle e_2, E, (\text{arg } l :: K), (e_1, E, (\text{fun } l :: K) \rangle, \text{ new } l \]
The Second Half: Provable Implementation Bounds

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

$m = \# \text{ of distinct variable names in } e$

in practice constant (will assume from now on)

* assumes implicit representation of result with sharing. For explicit representation, need to add $(|v|/p)$ term.
The Second Half: Provable Implementation 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 $\lambda$-calculus
(including constants)

$c \downarrow c; \quad 1, 1$  \hspace{1cm} (CONST)

$e_1 \downarrow c; \quad w_1, d_1 \quad e_2 \downarrow v; \quad w_2, d_2 \quad \delta(c, v) \downarrow v'$
\hspace{1cm} (APPC)

$e_1 e_2 \downarrow \nu'; \quad 1 + w_1 + w_2, \quad 1 + \max(d_1, d_2)$

$c_n = 0, \cdots, n, +, +_0, \cdots, +_n, <, <_0, \cdots, <_n, \times, \times_0, \cdots, \times_n, \cdots$ (constants)
The Parallel $\lambda$-calculus
(including constants)

\[ c \Downarrow c; \quad \begin{array}{c} 1,1 \end{array} \quad \text{(CONST)} \]

\[ e_1 \Downarrow c; \quad \begin{array}{c} w_1, d_1 \end{array} \quad e_2 \Downarrow v; \quad \begin{array}{c} w_2, d_2 \end{array} \quad \delta(c,v) \Downarrow v' \quad \text{(APPC)} \]

\[ e_1 e_2 \Downarrow v'; \quad \begin{array}{c} 1 + w_1 + w_2, \quad 1 + \max(d_1,d_2) \end{array} \]

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

The model we use in an introductory algorithms course at CMU
(almost).
A special case

**Corollary:** [FPCA95]: If $e \downarrow v; w,$ then $v$ can be calculated from $e$ on a RAM in $O(w \log m)$ time.
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)
fun qsort Empty = Empty
| qsort S =
  let val a = first S
    val S_1 = filter (fn x => x < a) S
    val S_2 = filter (fn x => x = a) S
    val S_3 = filter (fn x => x > a) S
  in
    append (qsort S_1) (append S_2 (qsort S_3))
  end
Qsort Complexity

Parallel partition
Parallel calls

Span = $O(\lg n)$

Work = $O(n \log n)$

Span = $O(\lg^2 n)$

A good parallel algorithm

Parallelism = $O(n/\log n)$
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)
Qsort Complexity

Parallel partition
Parallel calls

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

Work = $O(n \log n)$

All expected case

A good parallel algorithm

Span = $O(\log n)$

Parallelism = $O(n/\log n)$
The Parallel Speculative $\lambda$-calculus: cost model

Can apply the argument before it is fully computed, allows for pipelined parallelism

- Futures
- I-structures
The Parallel Speculative λ-calculus: cost model

\[ d \triangleright e \downarrow v; \ w, \ d', \ \hat{d} \]

Evaluate e starting at depth (time) \( d \),
returning value \( v \)
with work \( w \)
with “min” (available) depth \( d' \)
and “max” (completed) depth \( \hat{d} \)
The Parallel Speculative $\lambda$-calculus: Cost Model

\[ d > e \Downarrow v; \ w, \ d', \ \hat{d} \]

\[ e_1 \Downarrow \lambda x . e \quad e_2 \Downarrow v \]

\[ e[v/x] \Downarrow v' \]

\[ e_1 \quad e_2 \]

\[ d \quad d' \]

\[ \hat{d} \]
The Parallel Speculative \( \lambda \)-calculus: Cost Model

\[ d \triangleright e \Downarrow v; \ w, \ d', \ \hat{d} \]

\[ \hat{d} = 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3) \]
The Parallel Speculative $\lambda$-calculus: cost model

\[ d \triangleright \lambda x. e \downarrow \lambda x. e; 1, 1 + d, 1 + d \]

\[ d \triangleright (\lambda x. e; d') \downarrow \lambda x. e; 1, 1 + \max(d, d'), 1 + \max(d, d') \]

\[ d+1 \triangleright e_1 \downarrow \lambda x. e; w_1, d_1, \hat{d}_1 \]

\[ d+1 \triangleright e_2 \downarrow v; w_2, d_2, \hat{d}_2 \]

\[ d_1 \triangleright e[(v;d_2)/x] \downarrow v'; w_3, d_3, \hat{d}_3 \]

\[ d \triangleright e_1 e_2 \downarrow v'; 1 + w_1 + w_2 + w_3, d_3, 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3) \]
Provable Implementation Bounds

**Theorem** [POPL96]: If $v \triangleright e \triangleright v; w, d', \hat{d}$ then $v$ can be calculated from $e$ on a F&A CREW PRAM with $p$ processors in $O\left(\frac{w}{p} + \hat{d} \log p\right)$ time.
Modeling Space

\( \sigma, R \triangleright e \downarrow l, \sigma', s \)

Evaluate \( e \) with store \( \sigma \), and root set \( R \subseteq \text{dom}(\sigma) \)
returning label \( l \in \text{dom}(\sigma') \)
with updated store \( \sigma' \)
and space \( s \)
Modeling Space

\[ \sigma, R \triangleright \lambda x. e \downarrow l, \sigma[l \mapsto \lambda x. e], \text{space}(R \cup l) \]
where \( l \notin \text{dom}(\sigma) \)

\[ \sigma, R \cup \text{labels}(e_2) \triangleright e_1 \downarrow l_1, \sigma_1, s_1 \]

\[ \sigma_1, R \cup \{l\} \triangleright e_2 \downarrow l_2, \sigma_2, s_2 \]

\[ \sigma_1(l_1) = \lambda x. e \]

\[ \sigma_2, R \triangleright e[x / l_2] \downarrow l, \sigma_3, s_3 \]

\[ \sigma, R \triangleright e_1 \ e_2 \downarrow l, \sigma_3, \max(1 + s_1, 1 + s_2, s_3) \]
Provable Implementation Bounds

**Theorem** [ICFP96,06]: If \( \{l, \sigma, w, d, s\} \Downarrow e \) then \( \sigma(l) \) can be calculated from \( e \) on a RAM in \( O(s) \) space and on a CREW PRAM with \( P \) processors in \( O(s + pd\log p) \) space and

\[
O\left(\frac{w}{p} + d\log p\right)
\]
time.
Adding Functional Arrays: NESL

\[
\{ e_1 : x \text{ in } e_2 \mid e_3 \}
\]

\[
e'[v_i/x] \Downarrow v'_i; w_i, d_i \quad i \in \{1 \ldots n\}
\]

\[
\{ e' : x \text{ in } [v_1 \ldots v_n] \} \Downarrow [v'_1 \ldots v'_n]; 1 + \sum_{i=1}^{n} w_i, 1 + \max_{i=1}^{v} d_i
\]

Primitives:

\([-\!: 'a\text{ seq }\ast (\text{int},'a)\text{ seq }\rightarrow 'a\text{ seq}\]

\([q,n,x,i,a] \leftarrow [(0,d),(2,r),(0,i)]\]

\([i,n,r,i,a]\]

\(\text{elt, index, length}\)
Quicksort in NESL

function quicksort(S) =
  if (#S <= 1) then S
  else let
    a = S[elt(#S)];
    S1 = {e in S | e < a};
    S2 = {e in S | e = a};
    S3 = {e in S | e > a};
    R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];

Span = $O(\log n)$
Work = $O(n)$
Space = $O(n)$
Expected
Provable Implementation Bounds

Theorem: If $e \downarrow v; w,d,s$ 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 and $O(s + pd\log p)$ space.
Cache Efficient Algorithms

Ideal-cache model/IO Model

- CPU
- Fast memory
- Slow memory
- Block

$B$ : cost = 0
$B$ : cost = 1
Known Bounds

• Merge Sort: \( O\left( \frac{n}{B} \log_2 \frac{n}{M} \right) \)

• Optimal Sort: \( O\left( \frac{n}{B} \log_{(M/B)} \frac{n}{M} \right) \)

• Matrix Multiply: \( O\left( \frac{n^3}{B\sqrt{M}} \right) \)
Merging

Blocks (B = 4)

X

Y

Out
Lists

head of list

27 15 25 3 14 18 35 11 32

head of list

3 11 14 15 18 25 27 32 35
MergeSort

\[
\text{Total} = (\frac{kn}{B}) \log_2 \left( \frac{n}{2M} \right) = O\left( \frac{n}{B} \log_2 \left( \frac{n}{M} \right) \right)
\]

Requires careful memory allocation

Cache Cost

- \( kn/B \)
- \( kn/B \)
- \( \log_2 (n/2M) \)

Free

size = \( M \), just fits in cache
Functional MergeSort

```
fun mergeSort([]) = []
  | mergeSort([a]) = [a]
  | mergeSort(A) =
    let
      val (L,H) = split(A)
      fun merge([], B) = B
          | merge(A,[]) = A
          | merge((a::At), (b::Bt)) =>
            if (a < b) then !a :: merge(At, B)
            else !b :: merge(A, Bt)
    in
      merge(mergeSort(L),mergeSort(H))
    end
```
Our Model

- nursery ($\nu$)
  - size = M, not organized in blocks
  - allocations (writes)
  - sorted by time (live data only)
  - oldest

- read cache ($\rho$)
  - reads
  - cost = 0

- main memory ($\mu$)
  - cost = 1

- Rules similar to space model
Conclusions

$\lambda$-calculus good for modeling:

- sequential time (work)
- parallel time (nested parallelism)
- parallel time (futures)
- space
- arrays
- cache efficient algorithms