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

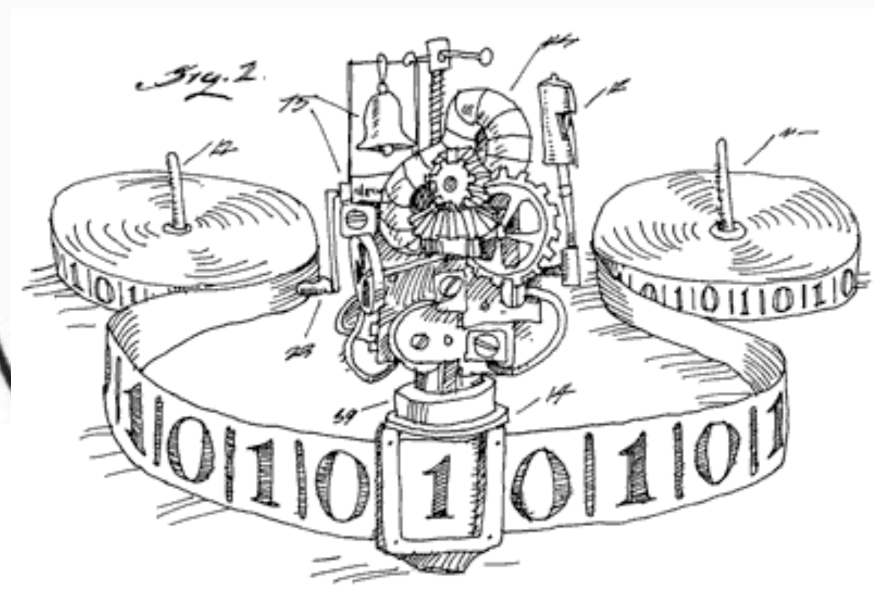
Guy Blelloch

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]

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 machines, called **machine models**."

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 machines, ~~called **machine**~~ **models**."

programming 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)

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

$$
\begin{aligned}
\text{LOGSPACE} &\subseteq \text{NLOGSPACE} \\
&\subseteq \text{P} \\
&\subseteq \text{NP} \\
&\subseteq \text{PSPACE} \\
&= \text{NPSPACE} \\
&\subseteq \text{EXPTIME} \\
&\subseteq \text{NEXPTIME} \\
&\subseteq \text{EXPSPACE} \\
&= \text{NEXPSPACE} \ldots
\end{aligned}
$$

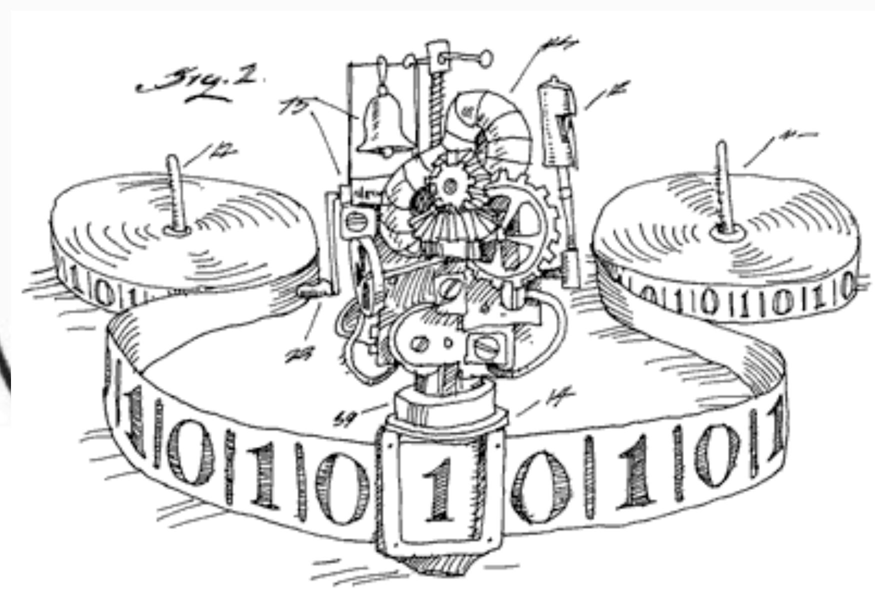
Parallel Machine Models

- 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 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

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 λ-Calculus [Plotkin 1975]
- Cost Semantics [Sands, Roe, ….]
- The lenient λ-Calculus [Roe 1991]
- λ-Calculus and linear speedups [SGM 2002]
- Various recent work [Martini, Dal Lago, Accattoli, …]
- Various work on “implicit computational complexity” (Leivant, Girard, Cook, …)

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

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

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

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?

The Parallel λ-calculus: cost model

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

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

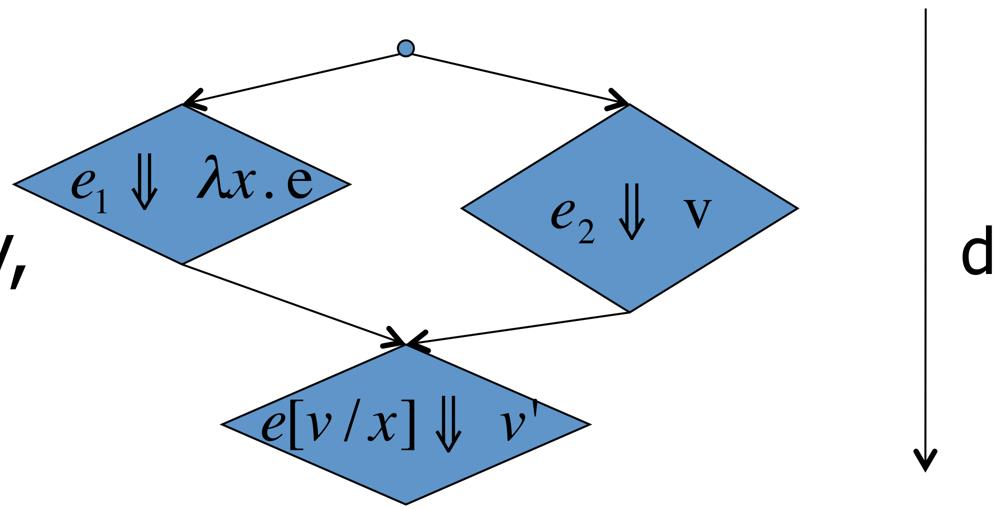
- **<u>Work</u>** (W): sequential work
- **<u>Span</u>** (D): parallel depth

The Parallel λ-calculus: cost model

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

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

Work adds
Span adds sequentially,
and max in parallel



The Parallel λ-calculus: cost model

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

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

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

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

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

$$K = \text{nil} \mid (\text{arg}\, l) :: K \mid (\text{fun}\ l) :: K$$

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

The Second Half: Provable Implementation Bounds

<u>**Theorem**</u> [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 = # 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 λ-calculus (including constants)

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

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

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

The Parallel λ-calculus (including constants)

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

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

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

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

A special case

<u>Corollary:</u> [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

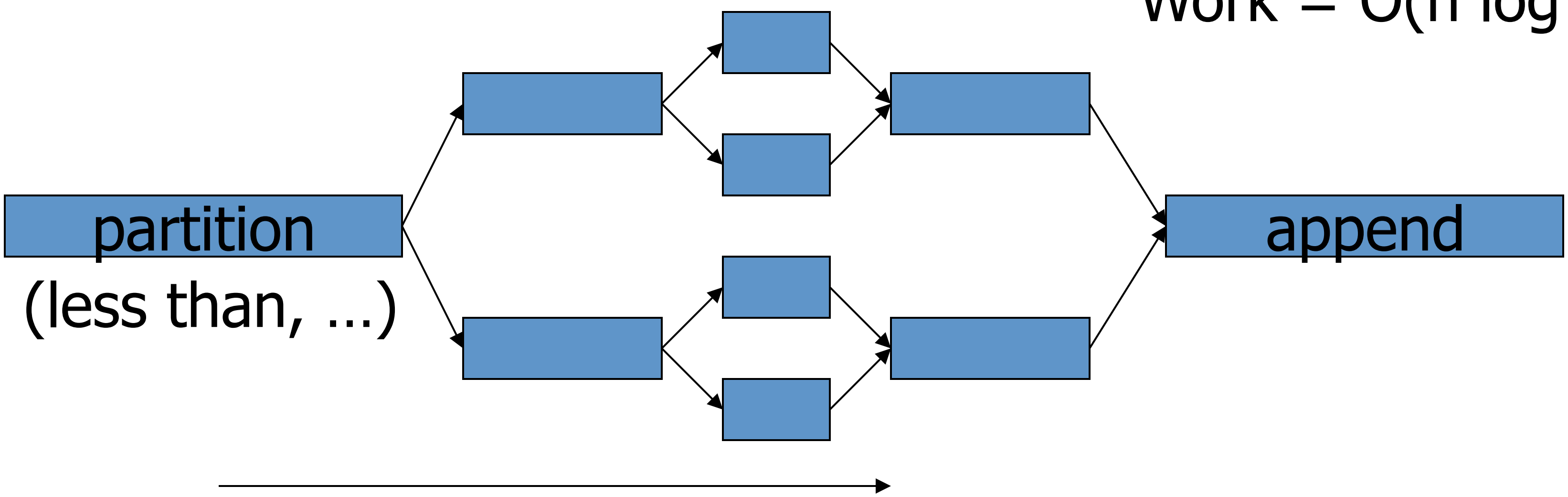
```
fun qsort [] = []
  | qsort S =
    let val a::_ = S
        val S1 = filter (fn x => x < a) S
        val S2 = filter (fn x => x = a) S
        val S3 = filter (fn x => x > a) S
    in
      append (qsort S1) (append S2 (qsort S3))
    end
```

Qsort Complexity

All bounds expected case over all inputs of size n

Sequential Partition
Parallel calls

Work = O(n log n)



(less than, ...)

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)
```

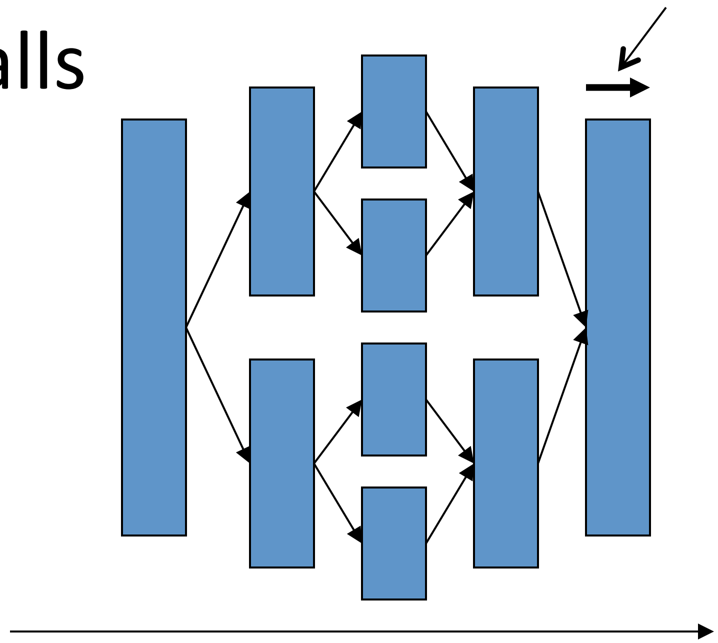
Tree Quicksort

```
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

Span = $O(\lg n)$



Work = $O(n \log n)$

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

A good parallel algorithm

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

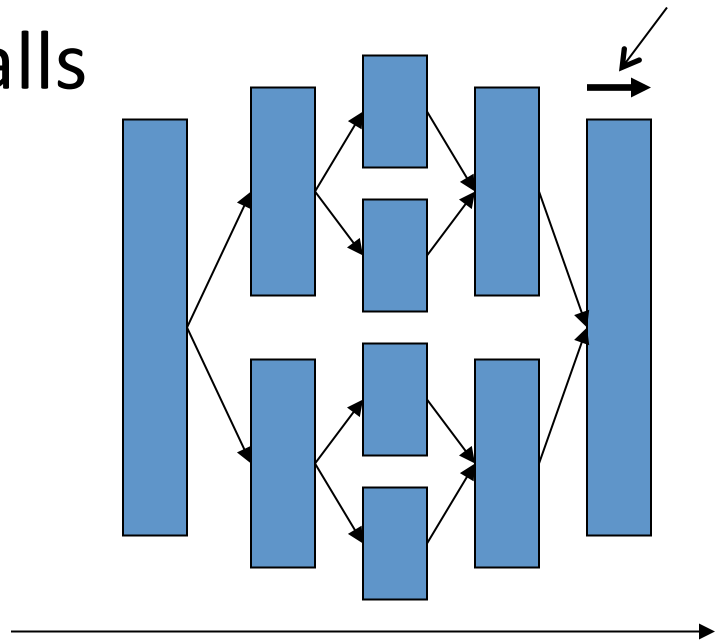
Tree Quicksort

```
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(lg n)



Work = O(n log n)

All expected case

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

A good parallel algorithm

Parallelism = O(n/log n)

The Parallel Speculative λ-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

$$d \triangleright e \Downarrow v;\ \mathrm{w},\ d',\ \hat{d}$$

Evaluate e starting at depth (time) *d,*

returning value v

with work w

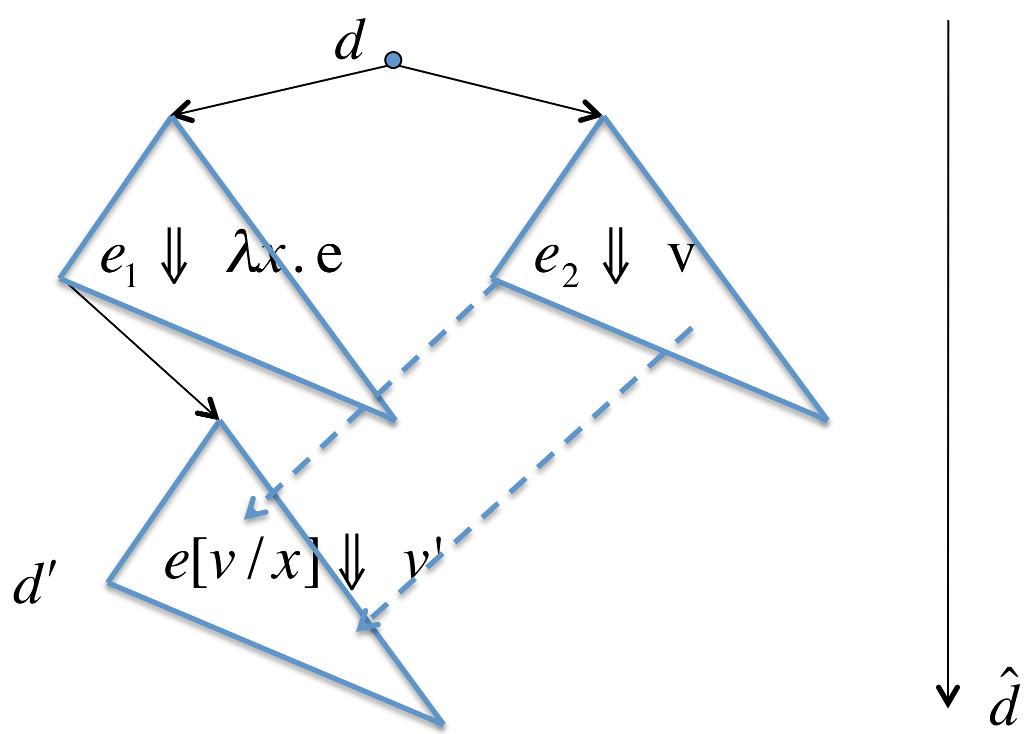
with “min” (available) detph d’

and “max” (completed) depth d hat

The Parallel Speculative λ-calculus: Cost Model

$$d \triangleright e \Downarrow v;\ \mathrm{w},\ d',\ \hat{d}$$

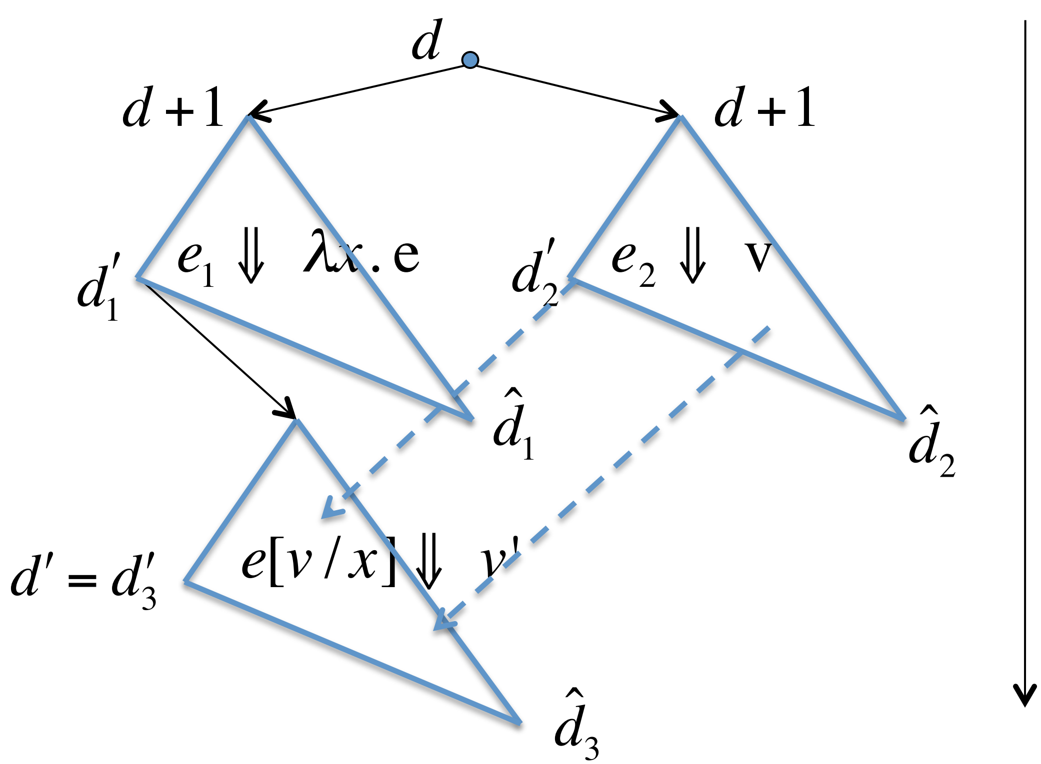
$e_1\ e_2$



The Parallel Speculative λ-calculus: Cost Model

$$d \triangleright e \Downarrow v;\ \mathrm{w},\ d',\ \hat{d}$$

$e_1\ e_2$



$$\hat{d} = 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3)$$

The Parallel Speculative λ-calculus: cost model

$$d \rhd \lambda x.\ e \Downarrow \lambda x.\ e;\ 1, 1+d, 1+d$$

$$d \rhd (\lambda x.\ e; d') \Downarrow \lambda x.\ e;\ 1,\ 1+\max(d,d'),\ 1+\max(d,d')$$

$$\frac{\begin{array}{c} d+1 \rhd e_1 \Downarrow \lambda x.\ e;\ w_1,\ d_1,\ \hat{d}_1 \\ d+1 \rhd e_2 \Downarrow v;\ w_2,\ d_2,\ \hat{d}_2 \\ d_1 \rhd e[(v;d_2)/x] \Downarrow v';\ w_3,\ d_3,\ \hat{d}_3 \end{array}}{d \underset{\lambda}{\rhd} e_1\ e_2 \Downarrow v';\ 1+w_1+w_2+w_3,\ d_3,\ 1+\max(\hat{d}_1,\hat{d}_2,\hat{d}_3)}$$

Provable Implementation Bounds

<u>Theorem</u> [POPL96]:If $0 \triangleright e \Downarrow v;\ w, d', \hat{d}$ 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 σ, and root set $R \subseteq dom(\sigma)$
returning label $l \in dom(\sigma')$
with updated store σ'
and space s

Modeling Space

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

$$\frac{\begin{array}{c}\sigma, R \cup \text{labels}(e_2) \triangleright e_1 \Downarrow l_1, \sigma_1, s_1 \\ \sigma_1, R \cup \{l\} \triangleright e_2 \Downarrow l_2, \sigma_2, s_2 \\ \sigma_1(l_1) = \lambda x.\ e \\ \sigma_2, R \triangleright e[x / l_2] \Downarrow l, \sigma_3, s_3\end{array}}{\sigma, R \triangleright e_1\ \ e_2 \Downarrow l, \sigma_3, \max(1 + s_1, 1 + s_2, s_3)}$$

Provable Implementation Bounds

<u>Theorem</u> [ICFP96,06]:If $\{\},\{\} \triangleright e \Downarrow l, \sigma, w, d, s$ 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 + pdlog p) space and $O\left(\frac{w}{p} + d \log p\right)$ time

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

$$\frac{e'[v_i / x] \Downarrow v_i'; w_i, d_i \qquad i \in \{1 \ldots n\}}{\{e' : x \text{ in } [v_1 \ldots v_n]\} \Downarrow [v_1' \ldots v_n']; \; 1 + \sum_{i=1}^{n} w_i, \; 1 + \max_{i=1}^{|v|} d_i}$$

Primitives:

<- : ‘a seq * (int,’a) seq -> ‘a seq

- **[q,n,x,i,a] <- [(0,d),(2,r),(0,i)]**
  [i,n,r,i,a]

[ICFP96]

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];
```

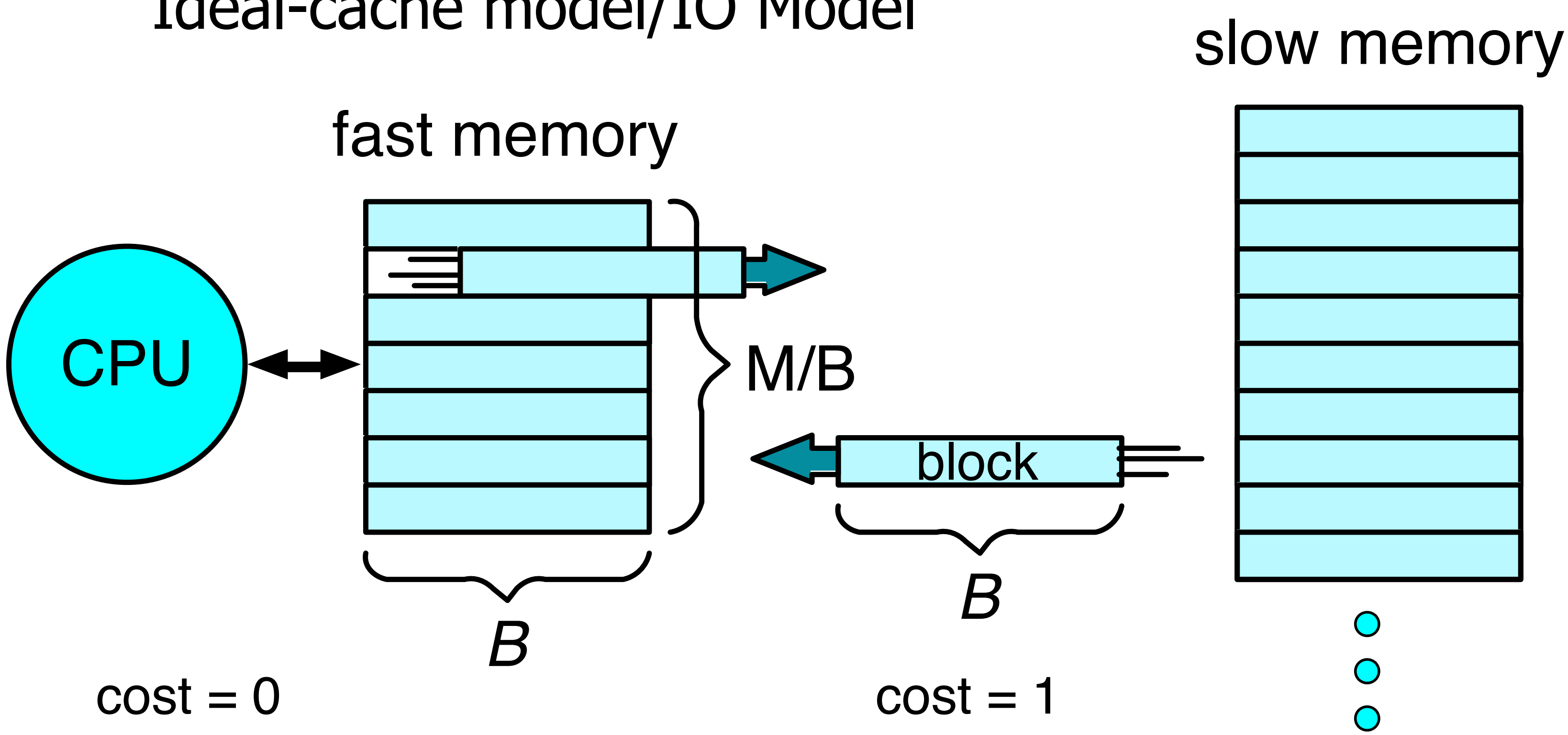
Span = **O(log n)**
Work = O(n)
Space = O(n)
Expected

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

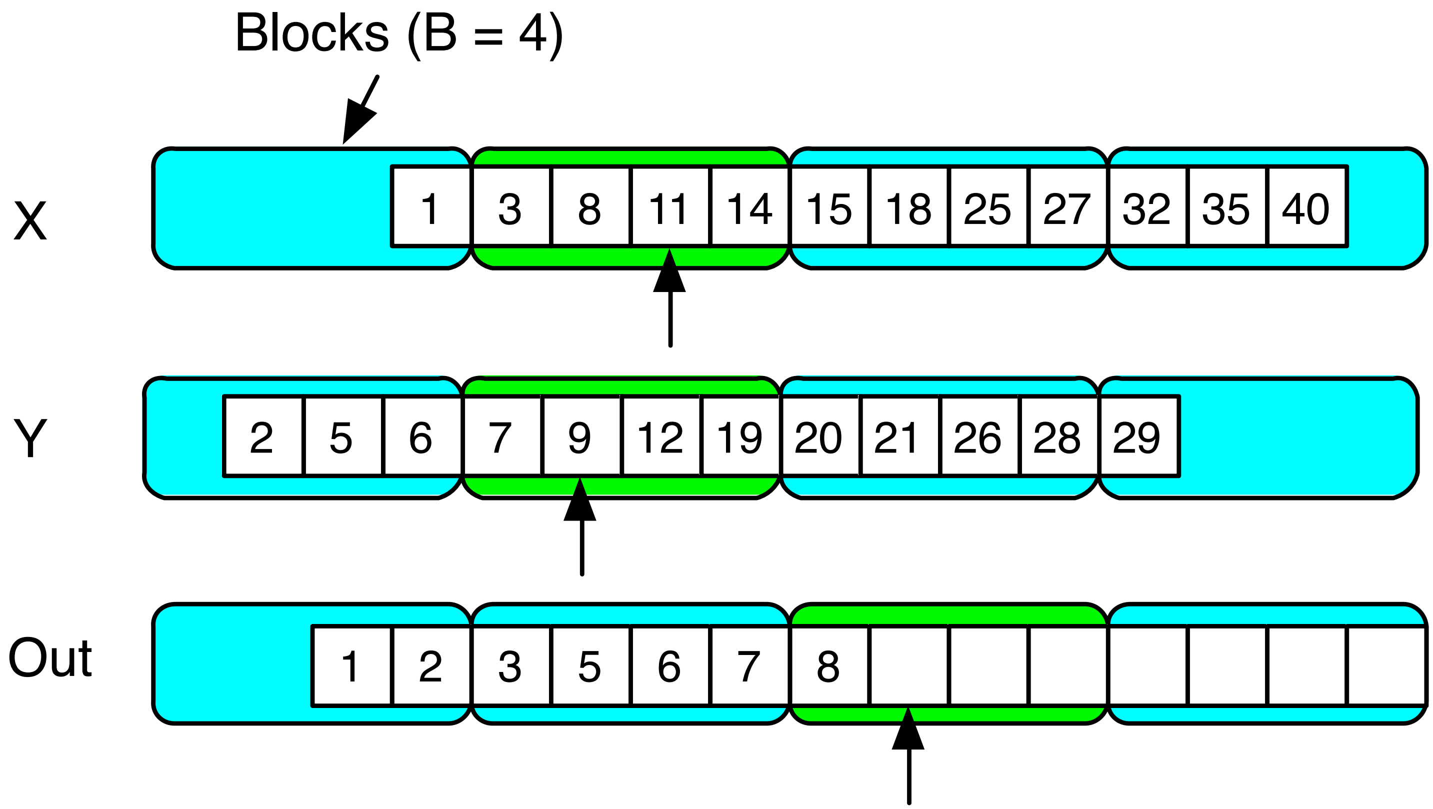
Ideal-cache model/IO Model



Known Bounds

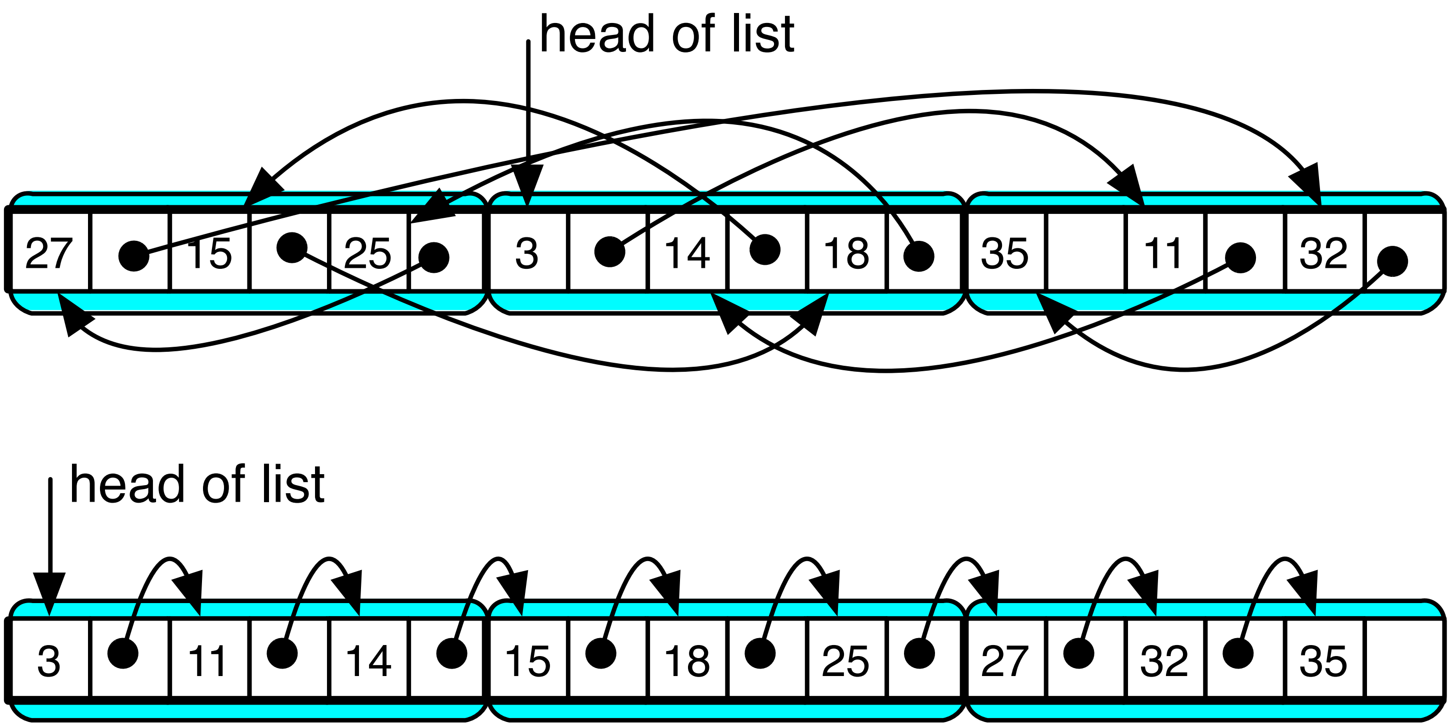
- Merge Sort: $O\left(\frac{n}{B}\log_2 \frac{n}{M}\right)$
- Optimal Sort: $O\left(\frac{n}{B}\log_{(M/B)} \frac{n}{M}\right)$
- Matrix Multiply: $O\left(\frac{n^3}{B\sqrt{M}}\right)$

Merging

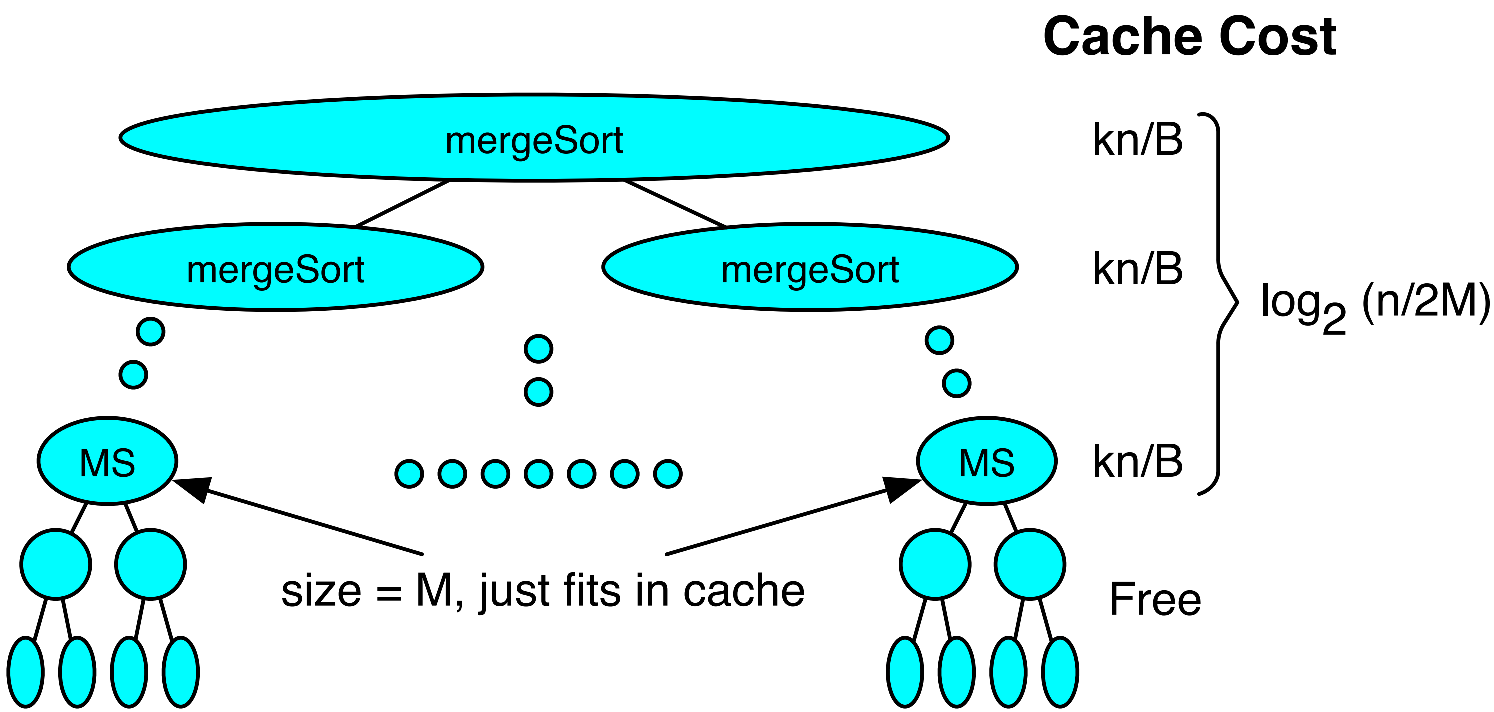


Blocks (B = 4)
X
1
3
8
11
14
15
18
25
27
32
35
40
Y
2
5
6
7
9
12
19
20
21
26
28
29
Out
1
2
3
5
6
7
8

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



Total $= (kn/B) \log_2 (n/2M)$

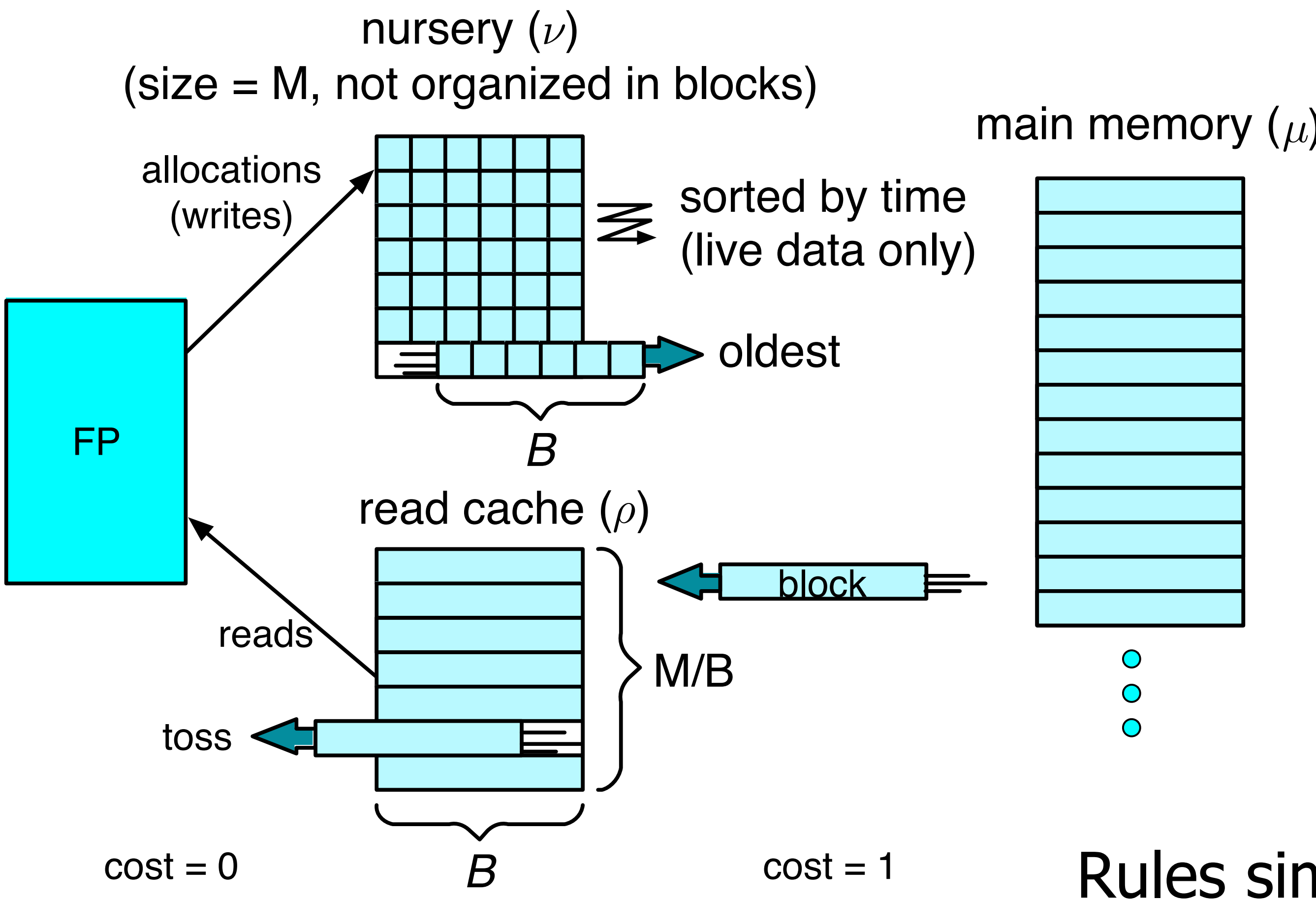
$= O(n/B \log_2 (n/M))$

Requires careful memory allocation

Functional MergeSort

```
fun mergeSort([]) = []
   | mergeSort([a]) = [a]
   | mergeSor(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



Rules similar to space model

Conclusions

λ-calculus good for modeling:

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