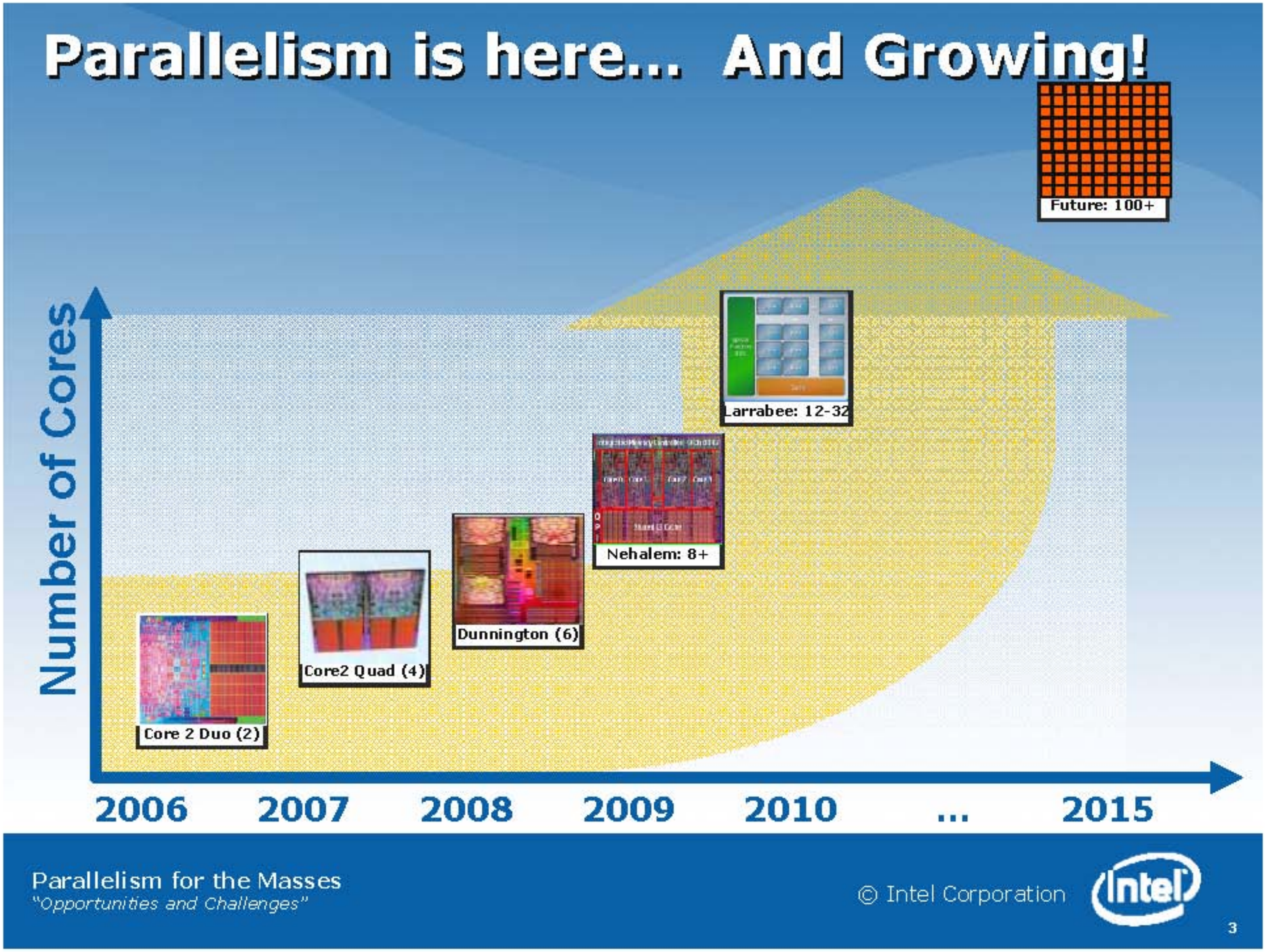
Functional Parallel Algorithms

Guy Blelloch
Carnegie Mellon University



Andrew Chien, 2008

Some benchmarks

Speedups on 32 cores (Dell Poweredge):

- Comparison sorting: 24x speedup
  - Sample sort (1 billion strings in 12 secs.)
- Minimum Spanning Tree: 17x speedup
  - Parallel Kruskal (1 billion edges in 8 secs.)
- K-nearest Neighbors: 14x speedup
  - Oct-tree (.1 billion points in 30 secs.)
- Delaunay Triangulation: 20x speedup
  - Incremental (.1 billion points in 48 secs.)
- Dictionary Insert+Lookup: 27x speedup
  - Hashing (1 billion strings in 6 secs.)

The State of Parallel Algorithms

- No accepted model by the algorithms/ complexity community.
- 136 papers Accepted to 2011 ACM/SIAM Symposium on Discrete Algorithms (SODA). 0 of them are about parallel algorithms.

Opportunity for the PL Community

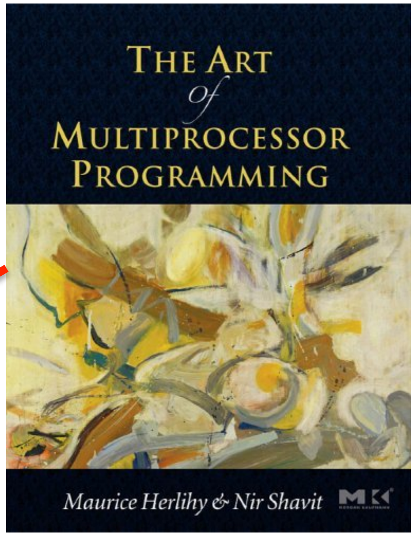
Reasons PL community can play a major role in how people will program and analyze parallel algorithms.

- Understand how to control effects
- Errors matter now
- Ease of programming matters
- Language based cost models
- “Parallel Thinking” is more natural.

Parallelism vs. Concurrency

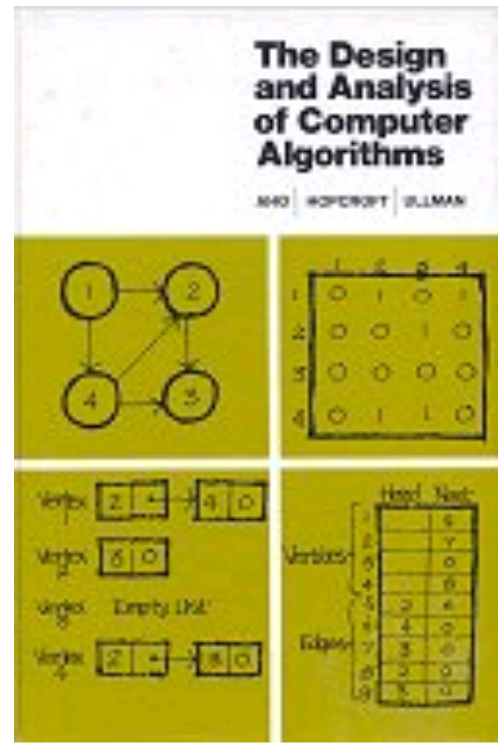
- Parallelism: using multiple processors/cores running at the same time. Property of the machine
- Concurrency: non-determinacy due to interleaving threads. Property of the application.

		Concurrency	
		sequential	concurrent
Parallelism	serial	Traditional programming	Traditional OS
	parallel	Deterministic parallelism	General parallelism



Quicksort from Aho-Hopcroft-Ullman

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



But….

We need a way to compare algorithms.

- How "parallel" is quicksort
- How does it compare to other sorting algorithms

We need a **formal cost model** so that we can make concrete claims.

Language Based Cost Models

A cost model based on the operational semantics

+

Provable implementation bounds

Call-by-value λ-calculus

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

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

The Parallel λ-calculus: cost model

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

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

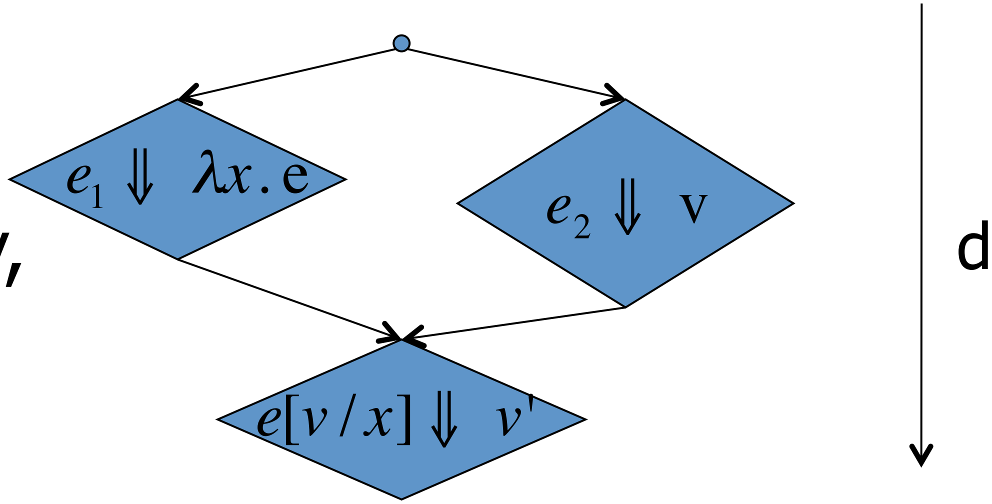
- **<u>Work</u>** (W): sequential work
- **<u>Span</u>** (D): parallel depth

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 implemented with O(log n) cost for integers up to n

The Parallel λ-calculus (constants)

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

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

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

The Parallel λ-calculus cost model

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

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

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

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

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

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.

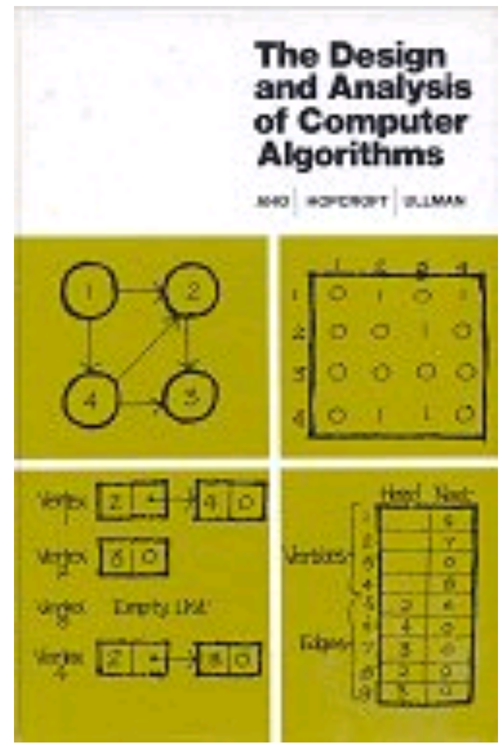
Can't really do better than: $\max\left(\frac{w}{p}, d\right)$

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

We refer to w/p as the parallelism.

Quicksort from Aho-Hopcroft-Ullman

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



Qsort on Lists

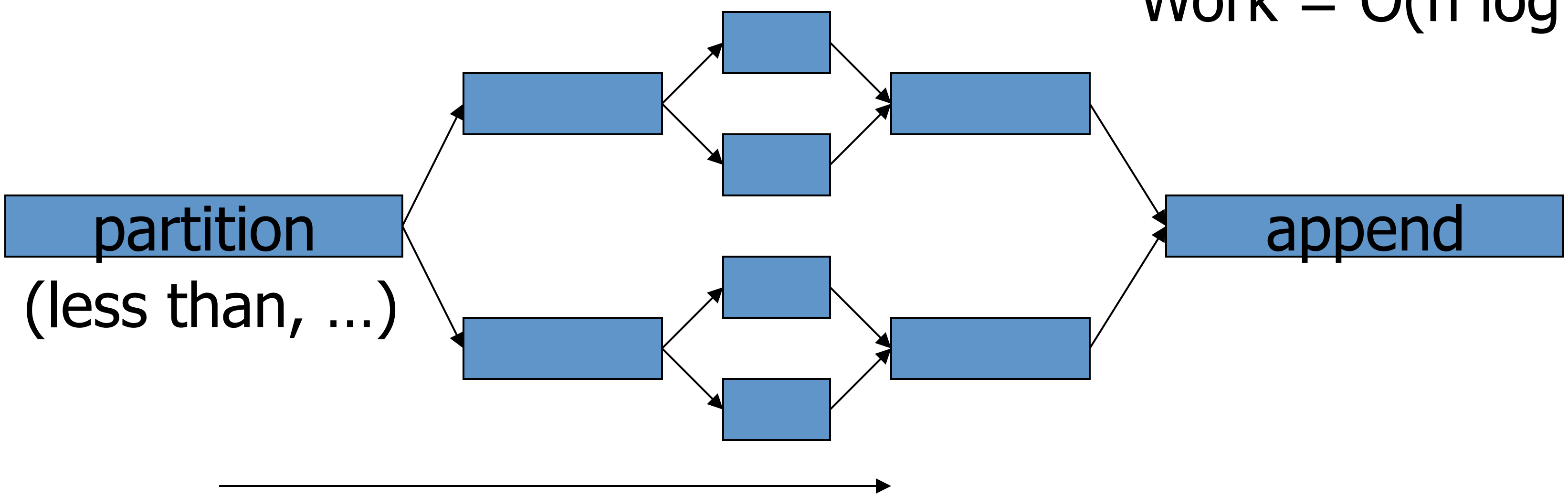
```
fun qsort [] = []
  | qsort S =
    let val a::_ = S
        val S1 = filter (fn x => x < a) S
        val S2 = filter (fn x => x = a) S
        val S3 = filter (fn x => x > a) S
    in
      append (qsort S1) (append S2 (qsort S3))
    end
```

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)
```

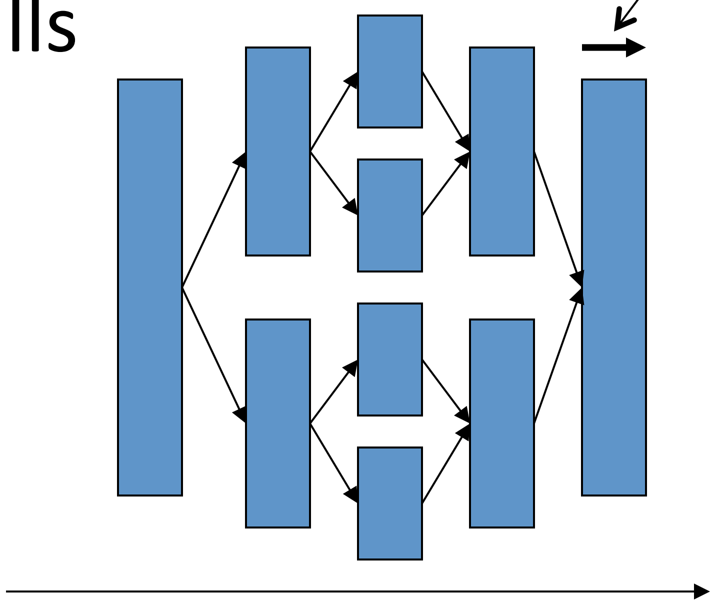
Tree Quicksort

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

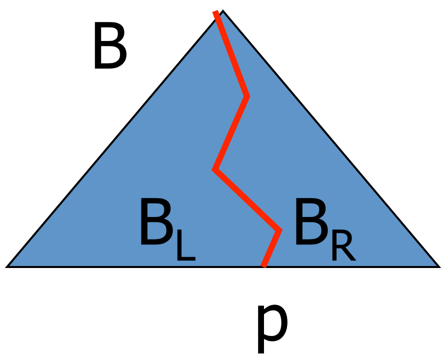
Example: Merging

```
Merge ([],l2) = l2
    | (l1,[]) = l1
    | (h1::t1, h2::t2) =
      if (h1 < h2) h1::Merge(t1,h2::t2)
      else h2::Merge(h1::t1,t2)
```

The Split Operation

```
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, Ll), R1) end;
```

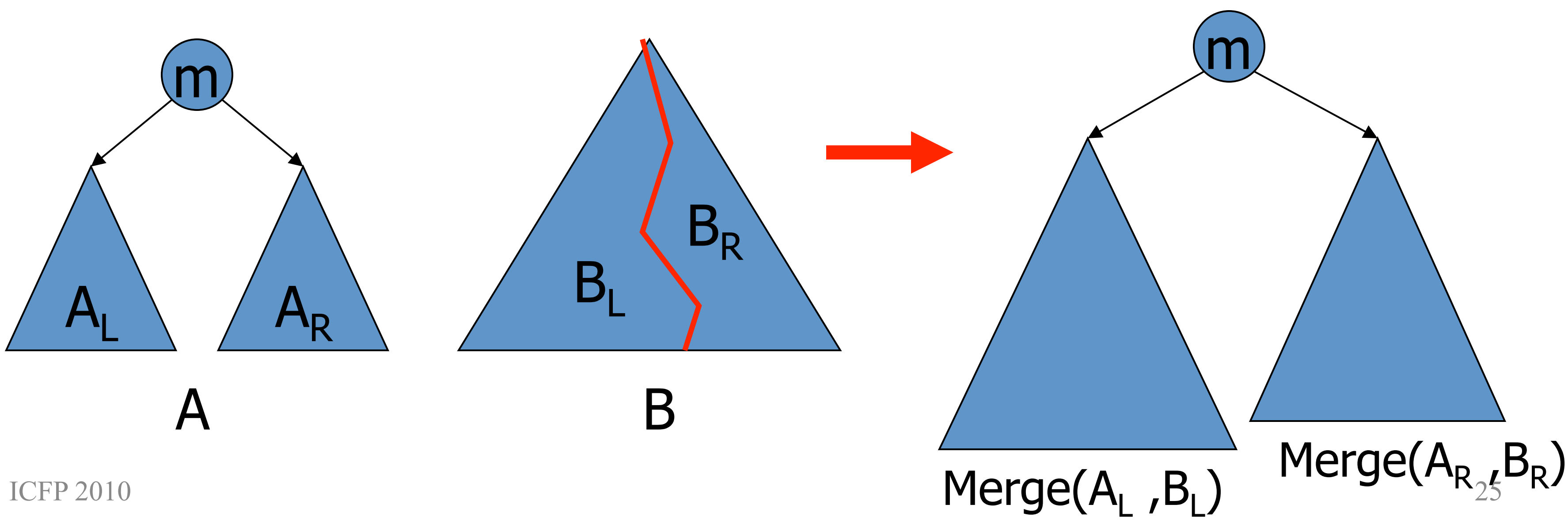


Merging

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

```
Merge(A,B) =
  let
    Node(A_L, m, A_R) = A
    (B_L ,B_R) = split(B, m)
  in
    Node(Merge(A_L,B_L), m, Merge(A_R,B_R))
```

Merge in parallel



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
```

- `[g,c,a,p] <- [(0,d),(2,f),(0,i)]`
  `[i,c,f,p]`

[ICFP95]

```
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.

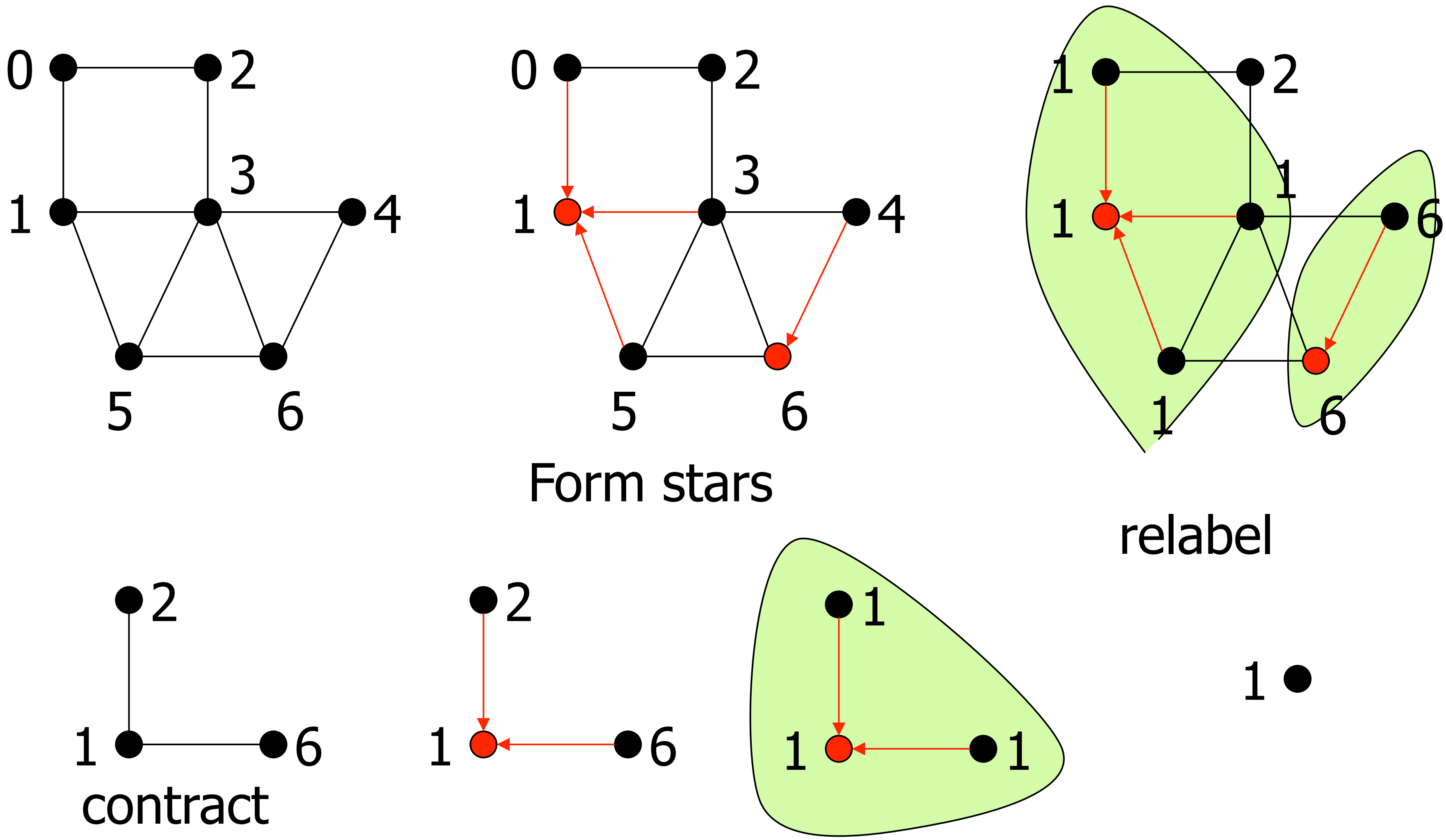
Interesting Side Note

Can implement hash tables so insertion of n elements takes:

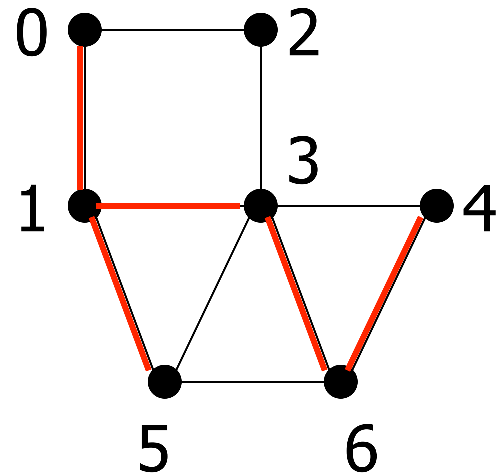
$W(n) = O(n)$ and $D(n) = O(\log n)$ expected case

Search takes $D(n) = W(n) = O(1)$ expected case

Example : Graph Connectivity



Example : Graph Connectivity



Edge List Representation:

```
Edges = [(0,1), (0,2), (2,3), (3,4), (3,5),
  (3,6), (1,3), (1,5), (5,6), (4,6)]

Hooks = [(0,1), (1,3), (1,5), (3,6), (4,6)]
```

Example : Graph Connectivity

L = Vertex Labels, E = Edge List

```
function connectivity(L, E) =
if #E = 0 then L
else let
  FL = {coinToss(.5) : x in [0:#L]};
  H = {(u,v) in E | Fl[u] and not(Fl[v])};
  L = L <- H;
  E = {(L[u],L[v]): (u,v) in E | L[u]\=L[v]};
in connectivity(L,E);
```

D = O(log n)

W = O(m log n)

Some Unfinished Problems

- How to take account of locality in a high-level way.
- Dealing properly with randomness
- Dealing properly with exceptions
- Efficient purely functional algorithms for many problems.

Summary

- Purely functional algorithms have several more advantages in parallel than sequentially.
- Programming-based cost models and implementation bounds could change the way people think about costs and open the door to all sorts of other “abstract” costs.
- Functional parallel algorithms are fun!!!!