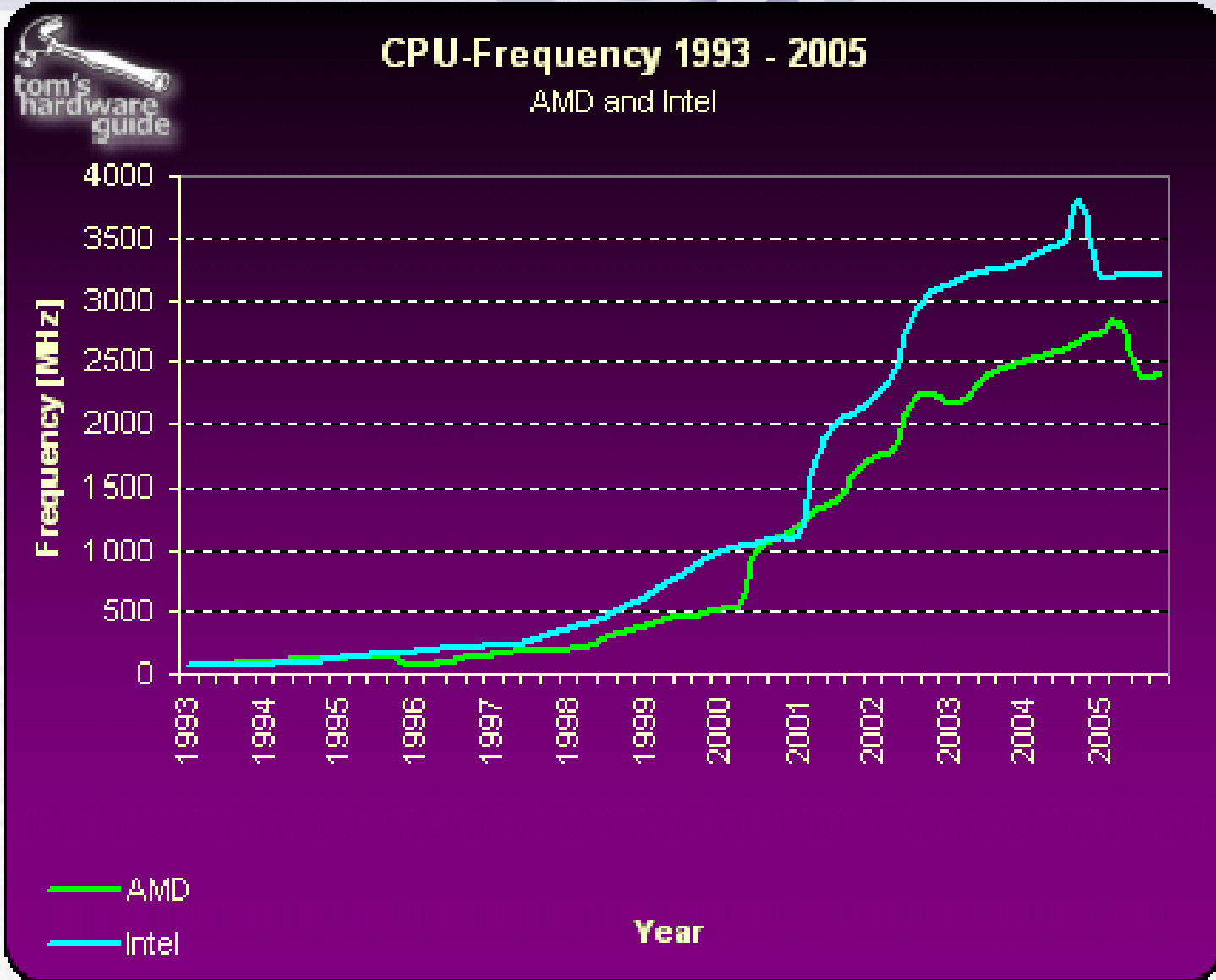




# Parallel Thinking\*

Guy Blelloch  
Carnegie Mellon University

\*PROBE as part of the Center for Computational Thinking

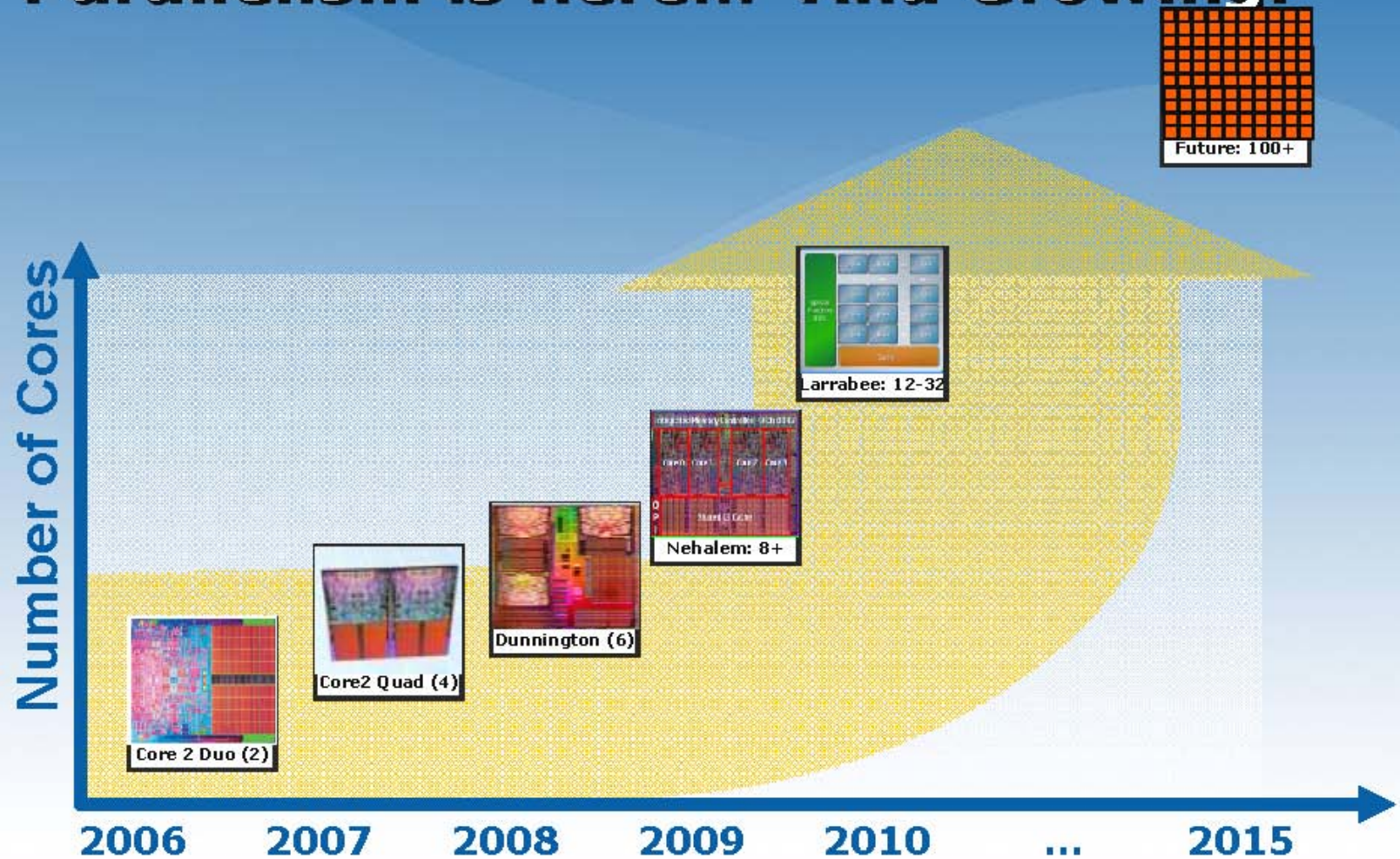


[http://www.tomshardware.com/2005/11/21/the\\_mother\\_of\\_all\\_cpu\\_charts\\_2005/](http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/)  
via Michael Scott



[http://www.tomshardware.com/2005/11/21/the\\_mother\\_of\\_all\\_cpu\\_charts\\_2005/](http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/)  
via Michael Scott

# Parallelism is here... And Growing!



Parallelism for the Masses  
*"Opportunities and Challenges"*

© Intel Corporation



# Parallel Thinking

## How to deal with parallelism/concurrency:

**Option I** : Minimize what users have to learn about parallelism. Hide parallelism in libraries which are programmed by a few experts

**Option II** : Teach parallelism as an advanced subject after and based on standard material on sequential computing.

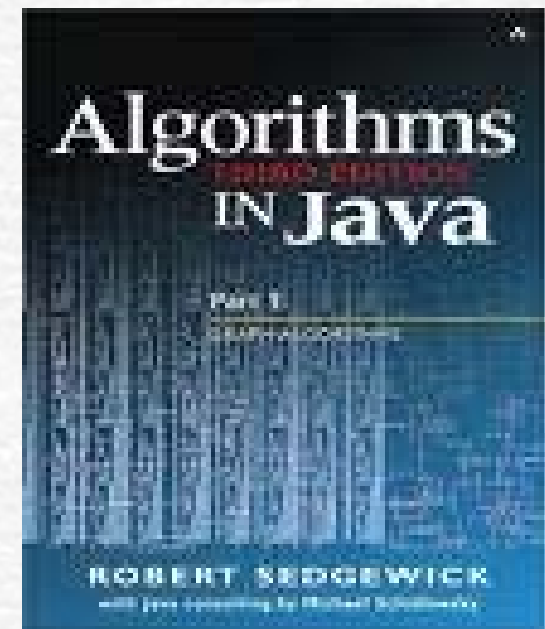
**Option III** : Teach parallelism from the start with sequential computing as a special case.

# Parallel Thinking

- Could it be that it is more natural to think about parallel algorithms than sequential algorithms?
- If done right could parallel programming be easier than sequential programming, or at least for most uses?
- Are we currently brainwashing students to think sequentially?
- **What are the core parallel ideas that all computer scientists should know?**

# Quicksort from Sedgwick

```
public void quickSort(int[] a, int left, int right) {  
    int i = left-1; int j = right;  
    if (right <= left) return;  
    while (true) {  
        while (a[++i] < a[right]);  
        while (a[right]<a[--j])  
            if (j==left) break;  
        if (i >= j) break;  
        swap(a,i,j); }  
    swap(a, i, right);  
    quickSort(a, left, i - 1);  
    quickSort(a, i+1, right); }  
}
```



# 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**;

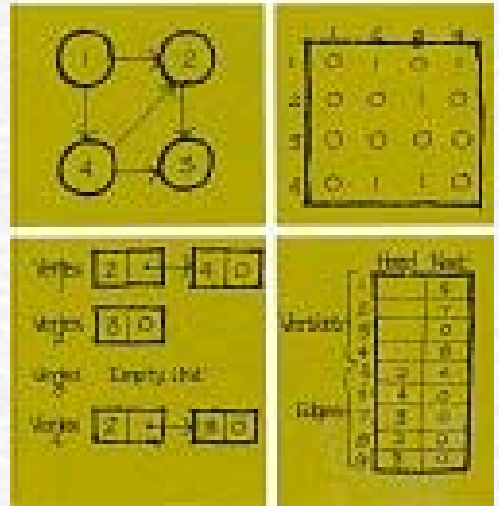
let **S**<sub>1</sub>, **S**<sub>2</sub> and **S**<sub>3</sub> be the sequences of elements in **S** less than, equal to, and greater than **a**, respectively;

return (QUICKSORT(**S**<sub>1</sub>) followed by **S**<sub>2</sub> followed by QUICKSORT(**S**<sub>3</sub>))

end

The Design and Analysis of Computer Algorithms

AWO | HOPCROFT | ULLMAN



# Lesson 1

- Natural parallelism is often lost in “low-level” implementations.
  - We need “higher level” languages
  - We need to revert back to the core ideas of an algorithm

# Quicksort in NESL

```
function quicksort(S) =  
  if (#S <= 1) then S  
  else let  
    a = S[rand(#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];
```

# Parallel selection

`{e in s | e < a};`

`S = [2, 1, 4, 0, 3, 1, 5, 7]`

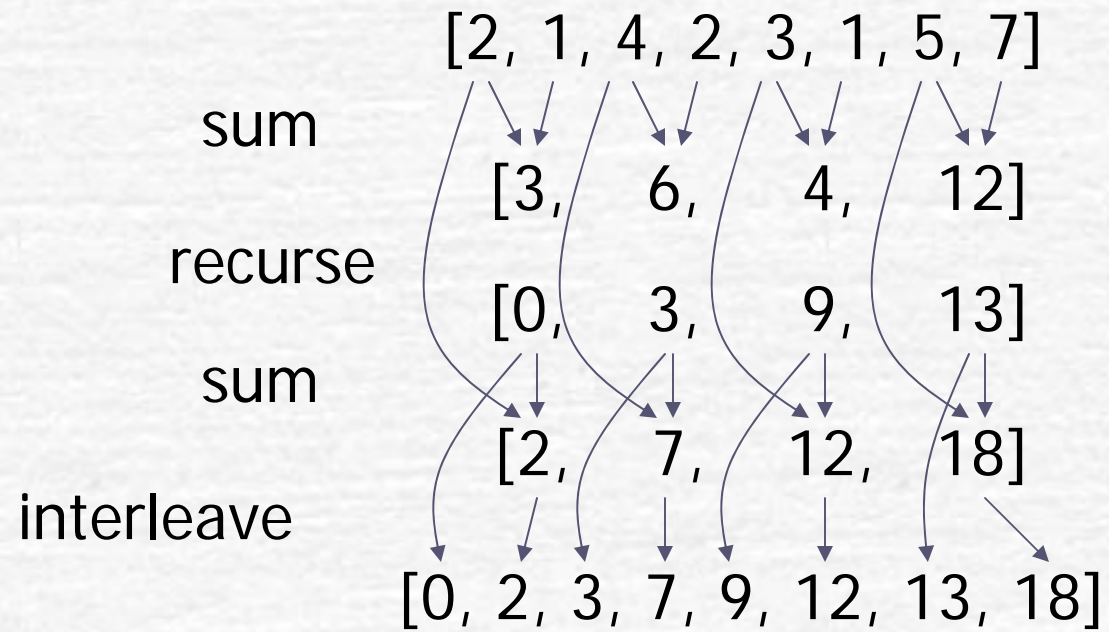
`F = S < 4 = [1, 1, 0, 1, 1, 1, 0, 0]`

`I = addscan(F) = [0, 1, 2, 2, 3, 4, 5, 5]`

where `F`

`R[I] = S = [2, 1, 0, 3, 1]`

# Scan



# Scan

```
function plusscan(A,op) =  
if (#A <= 1) then [0]  
else let  
  sums = {A[2*i] + A[2*i+1] : i in [0:#a/2]};  
  evens = scan(sums);  
  odds = {evens[i] + A[2*i] : i in [0:#a/2]};  
in interleave(evens,odds);,
```

A = [2, 1, 4, 2, 3, 1, 5, 7]

sums = [3, 6, 4, 12]

evens = [0, 3, 9, 13] (result of recursion)

odd = [2, 7, 12, 18]

result = [0, 2, 3, 7, 9, 12, 13, 18]

## Lessons 2 and 3

- Just because it seems sequential does not mean it is

And

- When in doubt recurse on a smaller problem

# Quicksort in Cilk++

```
seq quicksort(seq S) {  
    if (S.length < 2) return S;  
    double a = S[rand(S.length)];  
    seq S1,S2,S3;  
    cilk_spawn S1 = quicksort(lessThan(S,a));  
    cilk_spawn S2 = eqTo(S,a);  
    S3 = quicksort(grThan(S,a));  
    cilk_sync;  
    S1.append(S2.append(S3));  
}
```

# Quicksort in Cilk++

```
int cnt=0;

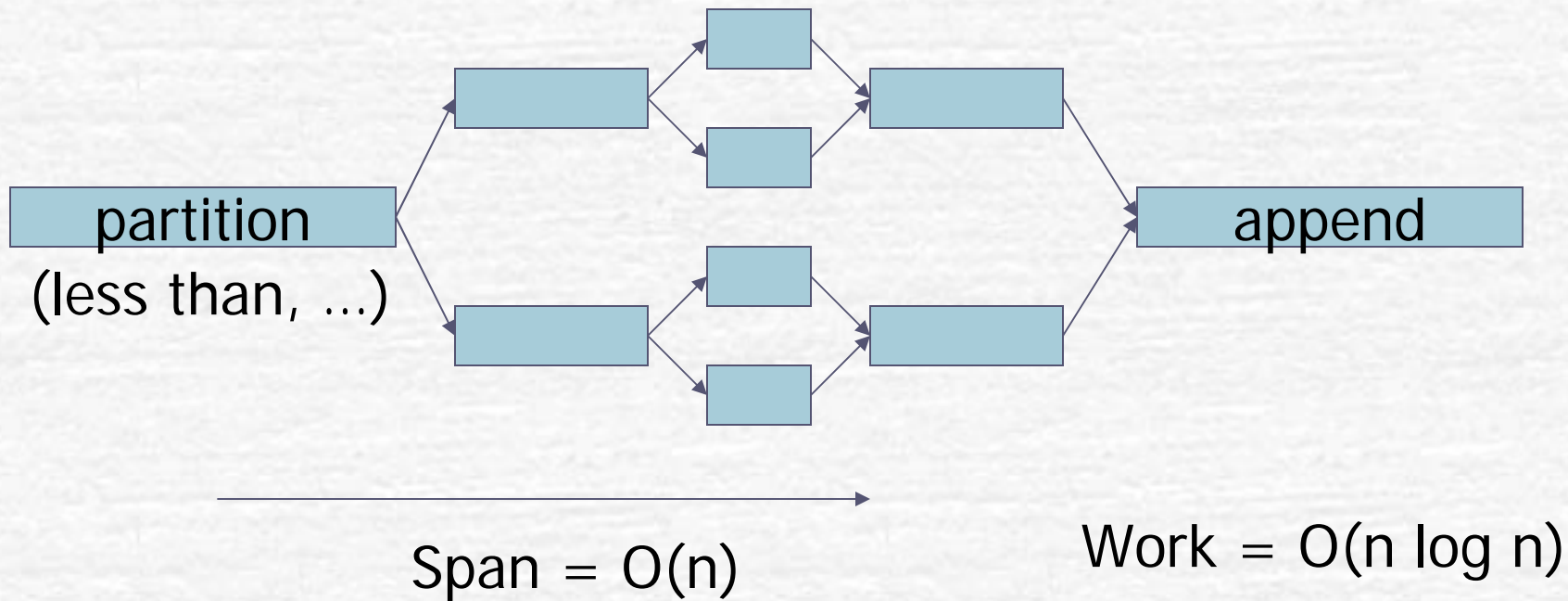
seq quicksort(seq S) {
    cnt++;
    if (S.length < 2) return S;
    double a = S[rand(S.length)];
    seq S1,S2,S3;
    cilk_spawn S1 = quicksort(lessThan(S,a));
    cilk_spawn S2 = eqTo(S,a);
    S3 = quicksort(grThan(S,a));
    cilk_sync;
    S1.append(S2.append(S3));
}
```

# Lesson 4

- Deterministic parallelism is important
  - Functional languages
  - Race detectors
  - Trying to make non-functional languages functional

# Complexity

## Sequential Partition and Append



# Quicksort in HPF

```
subroutine quicksort(a,n)
integer n,nless,less(n),greater(n),a(n)

if (n < 2) return

pivot = a(1)
nless = count(a < pivot)
less = pack(a, a < pivot)
greater = pack(a, a >= pivot)

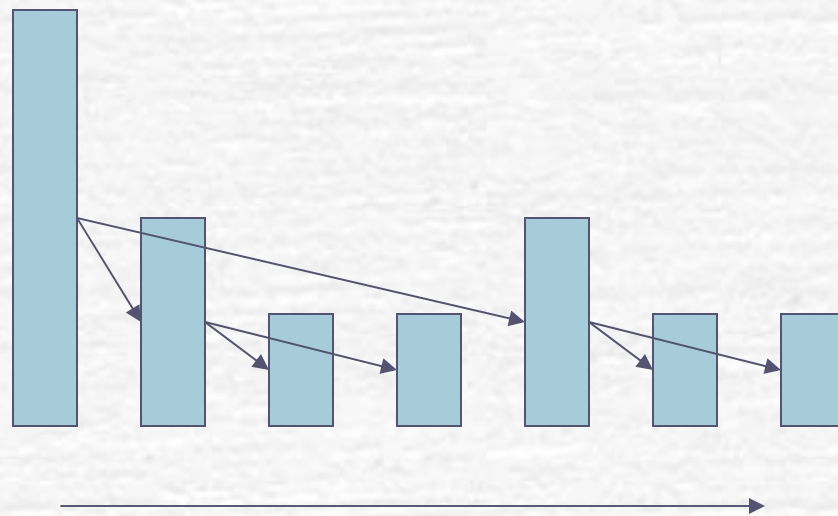
call quicksort(less, nless)
a(1:nless) = less

call quicksort(greater, n-nless)
a(nless+1:n) = less

end subroutine
```

# Complexity

Sequential calls to qsort

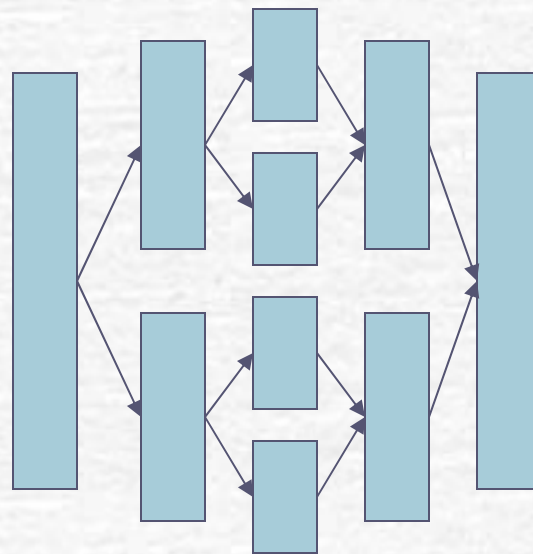


Span =  $O(n)$

Work =  $O(n \log n)$

# Complexity

## Parallel Partition and Append



$$\text{Work} = O(n \log n)$$

→  
 $\text{Span} = O(\lg^2 n)$

# Complexity in Nesl

Combining for parallel map:

$$\text{pexp} = \{\text{exp}(e) : e \text{ in } A\}$$

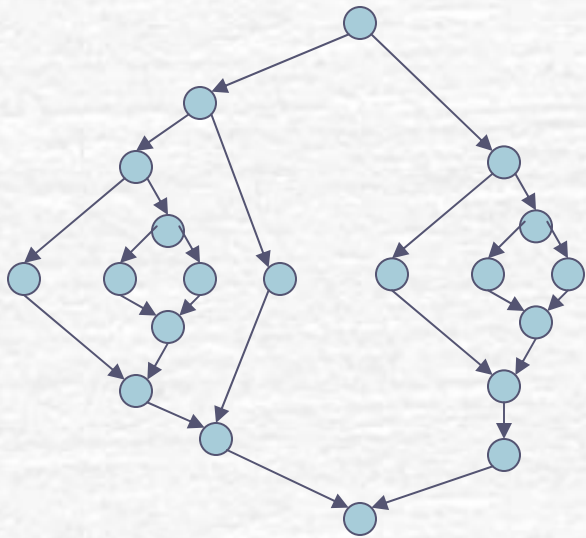
$$W_{\text{pexp}}(A) = \sum_{i=0}^{n-1} W_{\text{exp}}(A_i)$$

$$D_{\text{pexp}}(A) = \max_{i=0}^{n-1} D_{\text{exp}}(A_i)$$

Can prove runtime bounds for Various models:

$$T = O(W/P + D \log P)$$

# Generally for a DAG



- Any “greedy” schedule for a DAG with span (depth)  $D$  and work (size)  $W$  will complete in:

$$T < W/P + D$$

time

- But, different schedules have very different space characteristics

## Lessons 5, 6, 7 and 8

- Abstract cost models that are not machine based are important. Perhaps based in semantics of language.

and

- Work and span are reasonable measures

and

- Need to take advantage of both “data parallel” and “function parallel”

and

- Scheduling is important

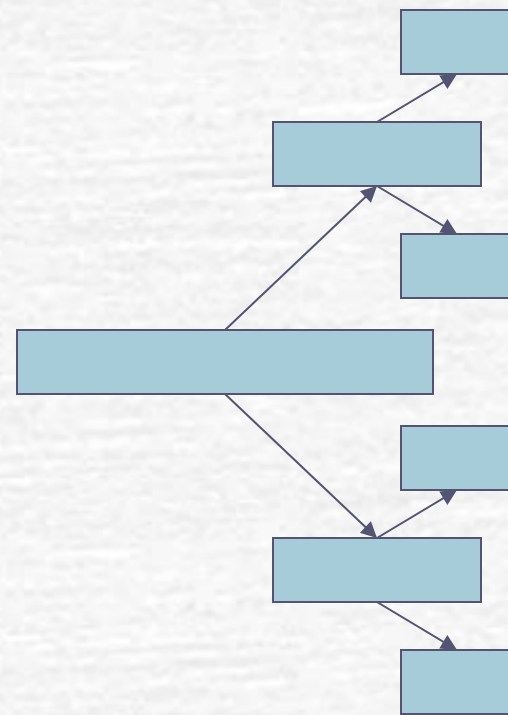
# Quicksort in Multilisp

```
(defun quicksort (L) (qs L nil))
```

```
(defun qs (L rest)
  (if (null L) rest
      (let ((a (car L))
            (L1 (filter (lambda (b) (< b a)) (cdr L)))
            (L3 (filter (lambda (b) (>= b a)) (cdr L))))
        (qs L1 (future (cons a (qs L3 rest)))))))
```

```
(defun filter (f L)
  (if (null L) nil
      (if (f (car L))
          (future (cons (car L) (filter f (cdr L)))
                  (filter f (cdr L))))))
```

# Quicksort in Multilisp (futures)



Work =  $O(n \log n)$

Span =  $O(n)$

## Lesson 9

- Pipelining might be useful but one should analyze span before using it blindly

# Example: Merging

```
Merge(nil,l2) = l2
```

```
Merge(l1,nil) = l1
```

```
Merge(h1::t1, h2::t2) =
```

```
  if (h1 < h2) h1::Merge(t1,h2::t2)
```

```
  else h2::Merge(h1::t1,t2)
```

What about in parallel?

# Merging

Merge(A, B) =

let

Node( $A_L, m, A_R$ ) = A

( $B_L, B_R$ ) = split(B, m)

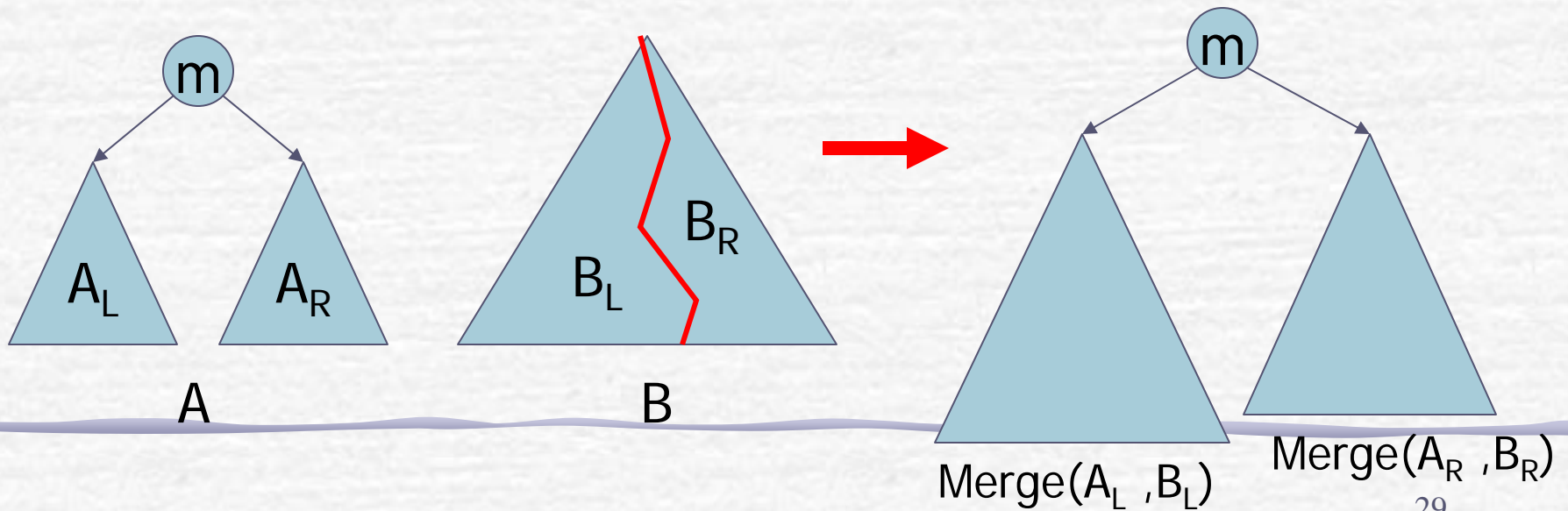
in

Node(Merge( $A_L, B_L$ ), m, Merge( $A_R, B_R$ ))

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

Work =  $O(n)$

Merge in parallel



# Merging with Futures

Merge(A,B) =

let

Node(A<sub>L</sub>, m, A<sub>R</sub>) = A

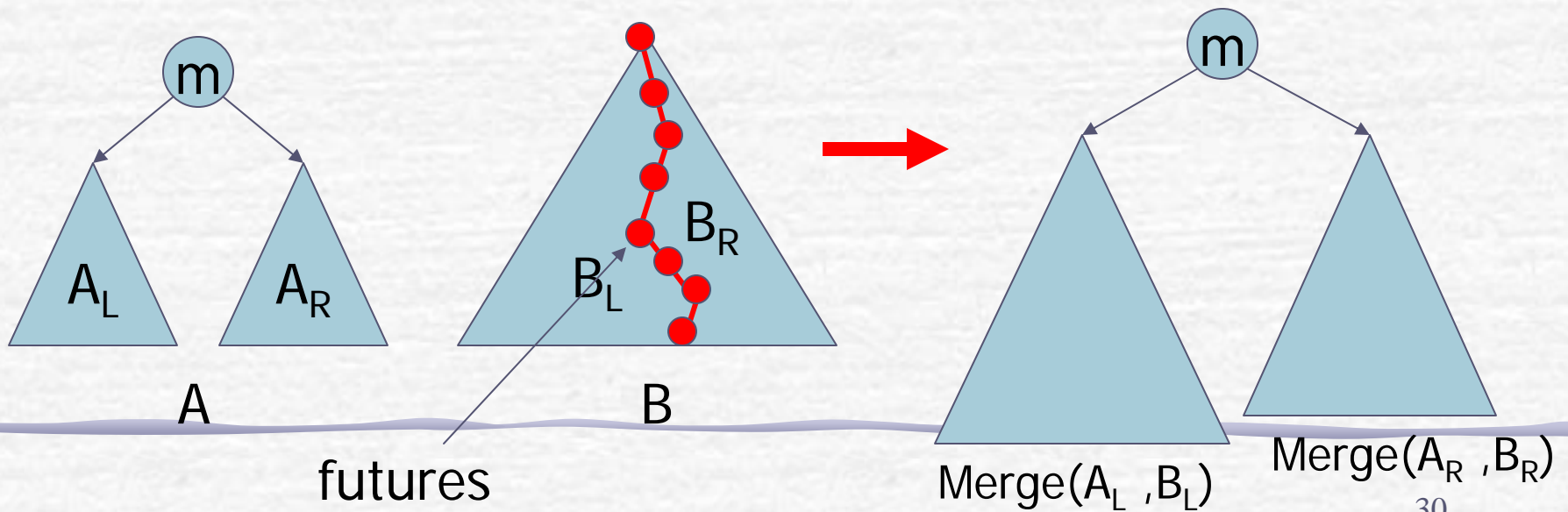
(B<sub>L</sub>, B<sub>R</sub>) = futureSplit(B, m)

in

Node(Merge(A<sub>L</sub>, B<sub>L</sub>), m, Merge(A<sub>R</sub>, B<sub>R</sub>))

Span = O(log n)

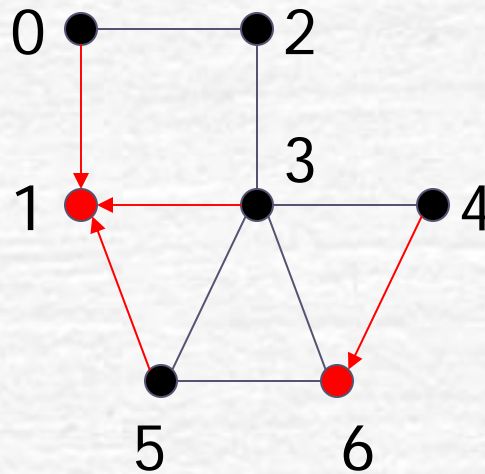
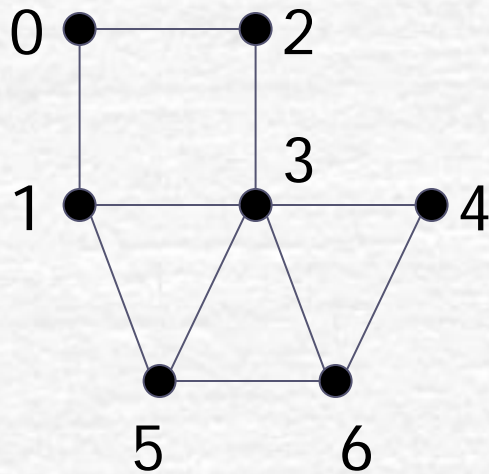
Work = O(n)



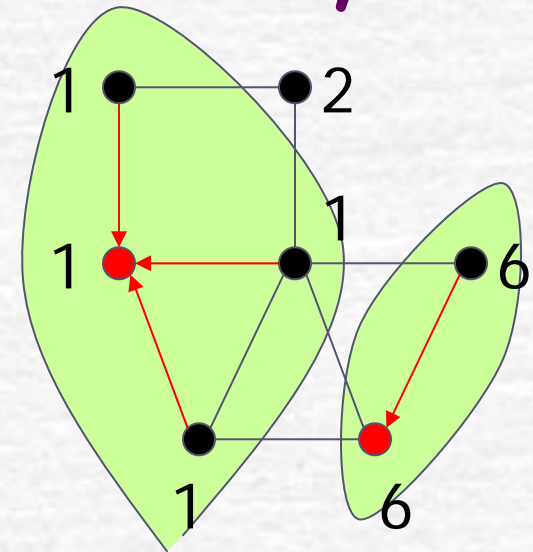
## Lessons 10, 11, and 9.5

- Divide and conquer even more useful in parallel than sequentially  
and
- Trees are better than lists for parallelism  
and
- (9.5) Pipelining is useful

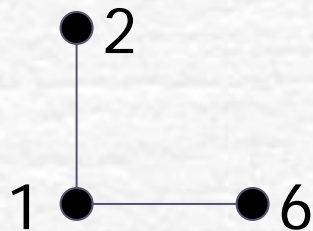
# Example : Graph Connectivity



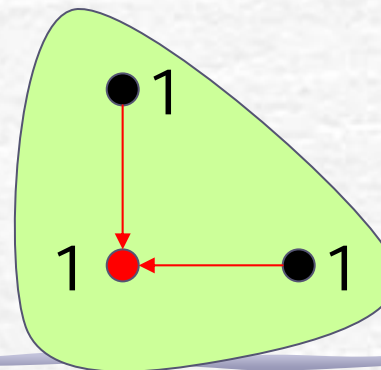
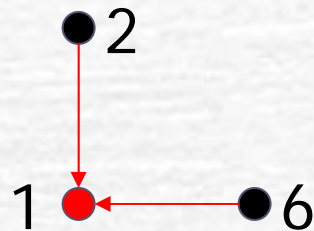
Form stars



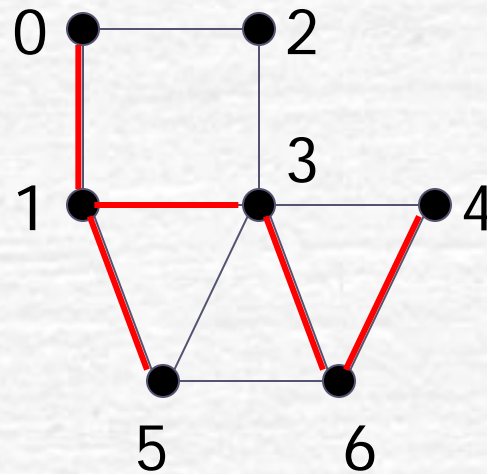
relabel



contract



# 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

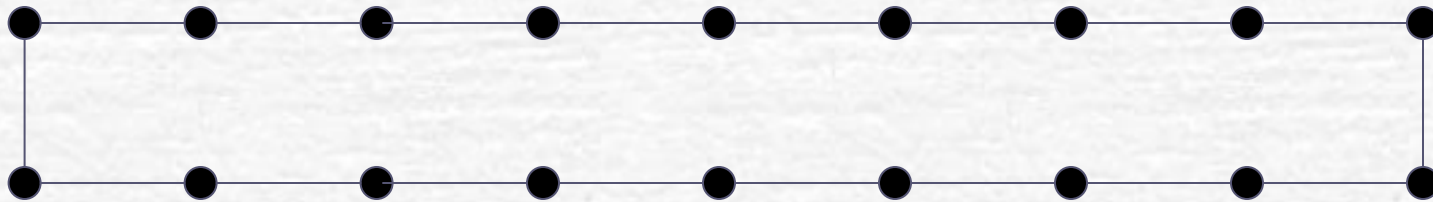
L = Vertex Labels, H = hooks E = Edge List

```
L = L <- H; /* Write tails to heads */
```

```
E = {(L[u],L[v]) /* update edge labels */  
      : (u,v) in E  
      | L[u]\=L[v]}; /* remove self edges */
```

# Example : Graph Connectivity

How do we find the stars?



The world looks the same from every node.  
We need to break symmetry.

# Example : Random Mate

$L$  = 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] \neq L[v]};  
  in connectivity(L,E);
```

$D = O(\log n)$

$W = O(m \log n)$

## Lessons 3, 12, 13

- ☞ When in doubt recurse on a smaller problem  
and
- ☞ We often need to break symmetry  
and
- ☞ Sampling is useful

## General:

1. Natural parallelism is often lost in “low-level” implementations.
2. Just because it seems sequential does not mean it is

## Model and Language:

4. Deterministic parallelism is important
5. Abstract cost models that are not machine based are important.
6. Work and span are reasonable measures
7. Need to take advantage of both “data” and “function” parallelism
8. Scheduling is important

## Algorithmic Techniques

3. When in doubt recurse on a smaller problem
9. Pipelining is useful, with care
10. Divide and conquer even more useful in parallel
11. Trees are better than lists for parallelism
12. We often need to break symmetry
13. Sampling is useful

# What else

## Non-deterministic parallelism:

- Pi-calculus
- Sequential consistency, serializability, linearizability, atomic primitives, locking techniques, transactions
- Races and race detection
- Lock and wait free algorithms

## Architectural issues

- Cache coherence, memory layout, latency hiding
- Network topology, latency vs. throughput
- ...

# Conclusions/Questions

On round 3 of parallel machines is it time to carefully consider what to teach about parallelism?

Relatively small set of core concepts.