Parallel Thinking[*]

Guy Blelloch

[*]Part of Center for Computational Thinking

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 subjet after and based on standard material on sequential computing.

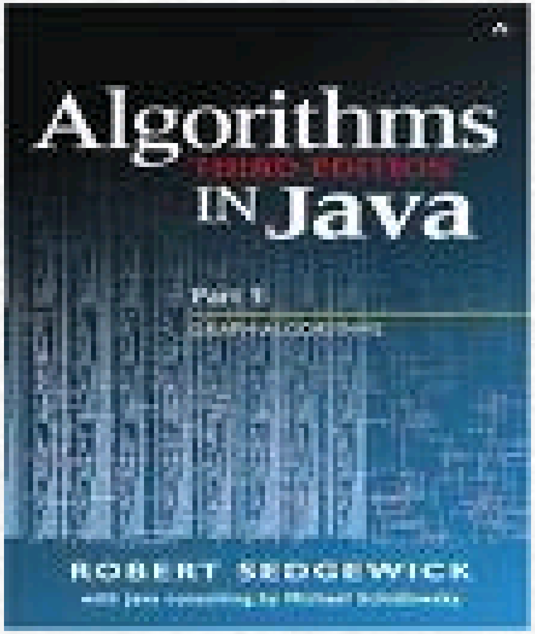
Option III : Teach parallelism from the start with sequential computing as a special case.

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 CS undergraduates should know?**

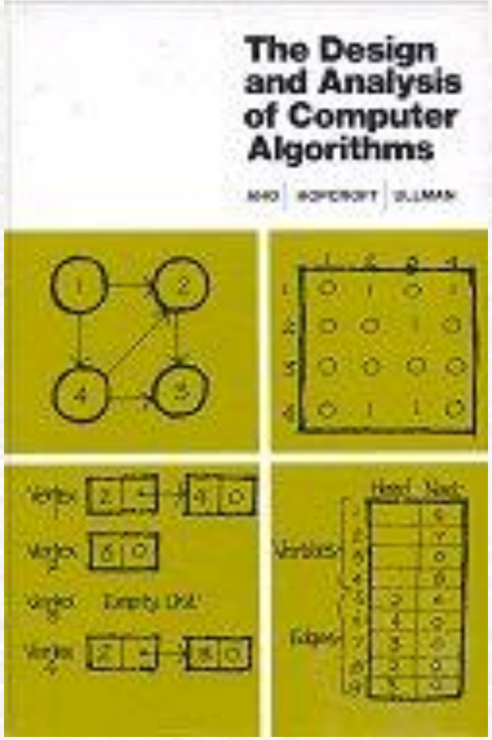
Quicksort from Sedgewick

```
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); }
```


Algorithms
IN Java
ROBERT SEDGEWICK

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 $S_1$, $S_2$ and $S_3$ be the sequences of
          elements in S less than, equal to,
          and greater than a, respectively;
      return (QUICKSORT($S_1$) followed by $S_2$
         followed by QUICKSORT($S_3$))
end
```



The Design
and Analysis
of Computer
Algorithms

AHO | HOPCROFT | ULLMAN

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 $S_1$, $S_2$ and $S_3$ be the sequences of
          elements in S less than, equal to,
          and greater than a, respectively;
     return (QUICKSORT($S_1$) followed by $S_2$
        followed by QUICKSORT($S_3$))
end
```

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];
```

Quicksort in X10

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

Parallel selection

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

$S = [2, 1, 4, 0, 3, 1, 5, 7]$
$F = S < 4 = [1, 1, 0, 1, 1, 1, 0, 0]$
$I = \mathrm{addscan}(F) = [0, 1, 2, 2, 3, 4, 5, 5]$

where F
    $R[I] = S = [2, 1, 0, 3, 1]$

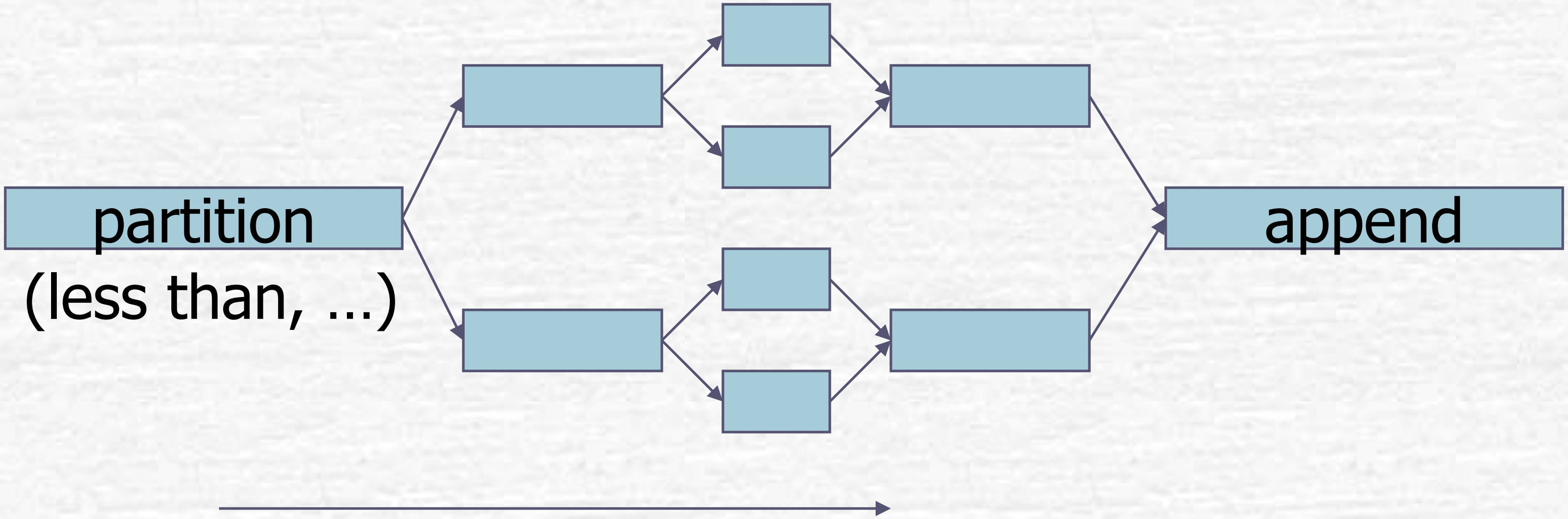
Scan

```
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]

Complexity

Sequential Partition and Append



partition

(less than, …)

append

Span = O(n)

Work = O(n log n)

Complexity in Nesl

Combining for parallel map:

```
pexp = {exp(e) : e in A}
```

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

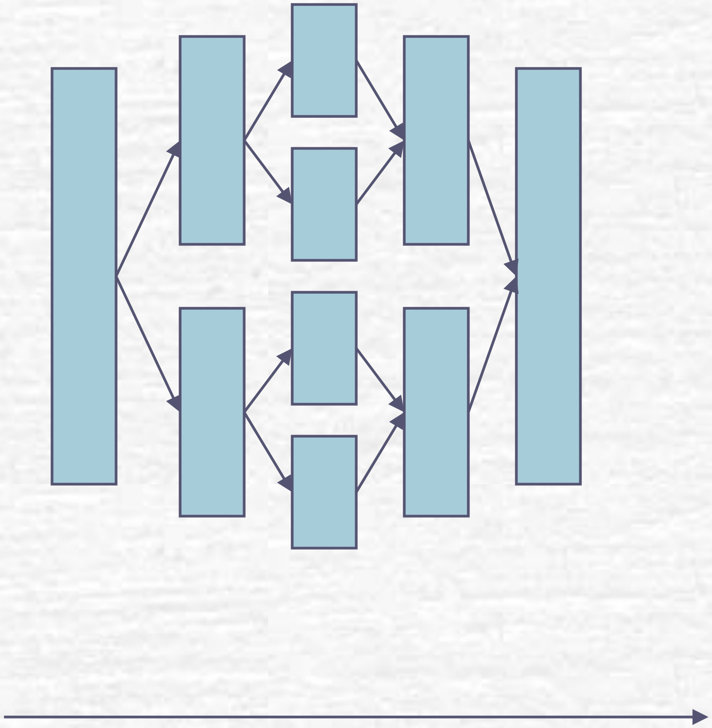
$$D_{\text{pexp}}(A) = \max_{i=0}^{n-1} D_{\text{exp}}(A_i)$$

Can prove runtime bounds for Various models:

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

Complexity

Parallel Partition and Append

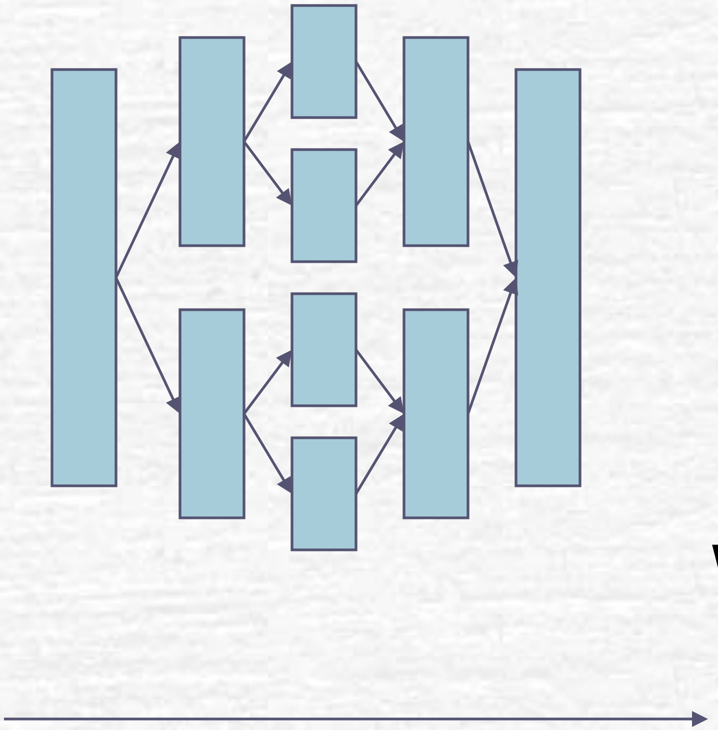


Work = $O(n \log n)$

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

Complexity

Parallel Partition and Append



Can add cache performance to the costs

Work = O(n log n)

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

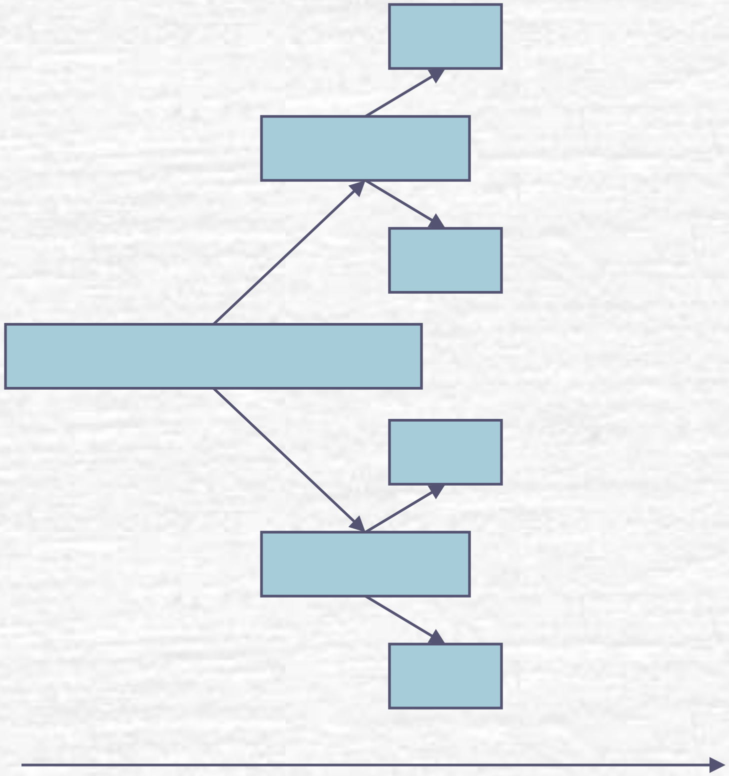
Quicksort in Multilisp (futures)

```
(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)))))
```

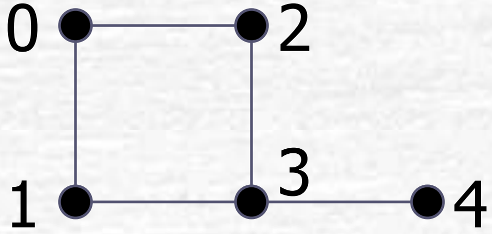
Quicksort in Multilisp (futures)



Work = O(n log n)

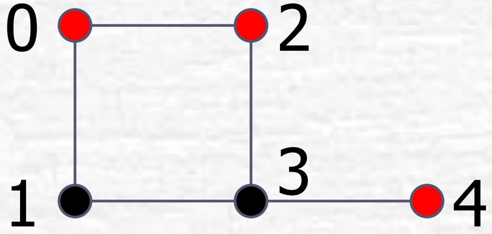
Span = O(n)

Example : Graph Connectivity

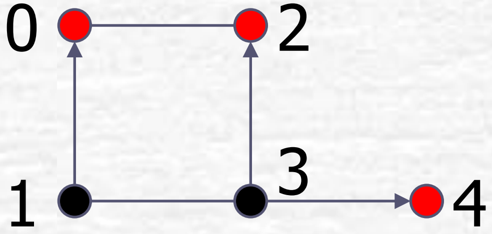


Edge List Representation:

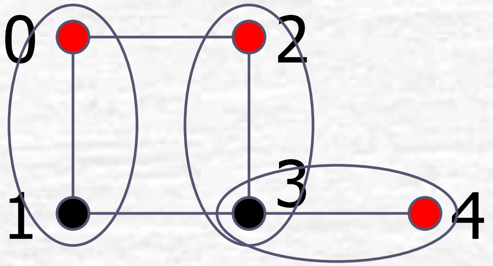
```
[(0,1), (0,2), (2,3), (3,4), (1,3),
 (1,0), (2,0), (3,2), (4,3), (3,1)]
```



Randomly flip coins



Every edge link
From black to red



Join

Example : Graph Connectivity

L = Vertex Labels, E = Edge List

```
function randomMate(L, E) =
if #E = 0 then L
else let
  FL = {randBit(.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 randomMate(L,E);
```

D = O(log n)
W = O(m log n)

Back to Parallel Thinking

- What are the core ideas that all undergraduates should know?

Concurrency vs. Parallelism

Some tasks are inherently **concurrent** (an OS, web server, multiuser game, sensor network).

Other tasks only use **parallelism** for efficiency (fft, nbody, frame rendering, shortest-path, mpeg-decode, speech understanding…)

Most applications will consists of both, but they should be separated and for the later we should use **deterministic semantics**.

- Longevity and composability
- reasoning about correctness
- Debugging

Proposal

- Develop a document that outlines what we think all undergraduates should know, and perhaps also what many should know.
  - What will still be useful 20 years from now?
  - What topics cover multiple areas or applications?