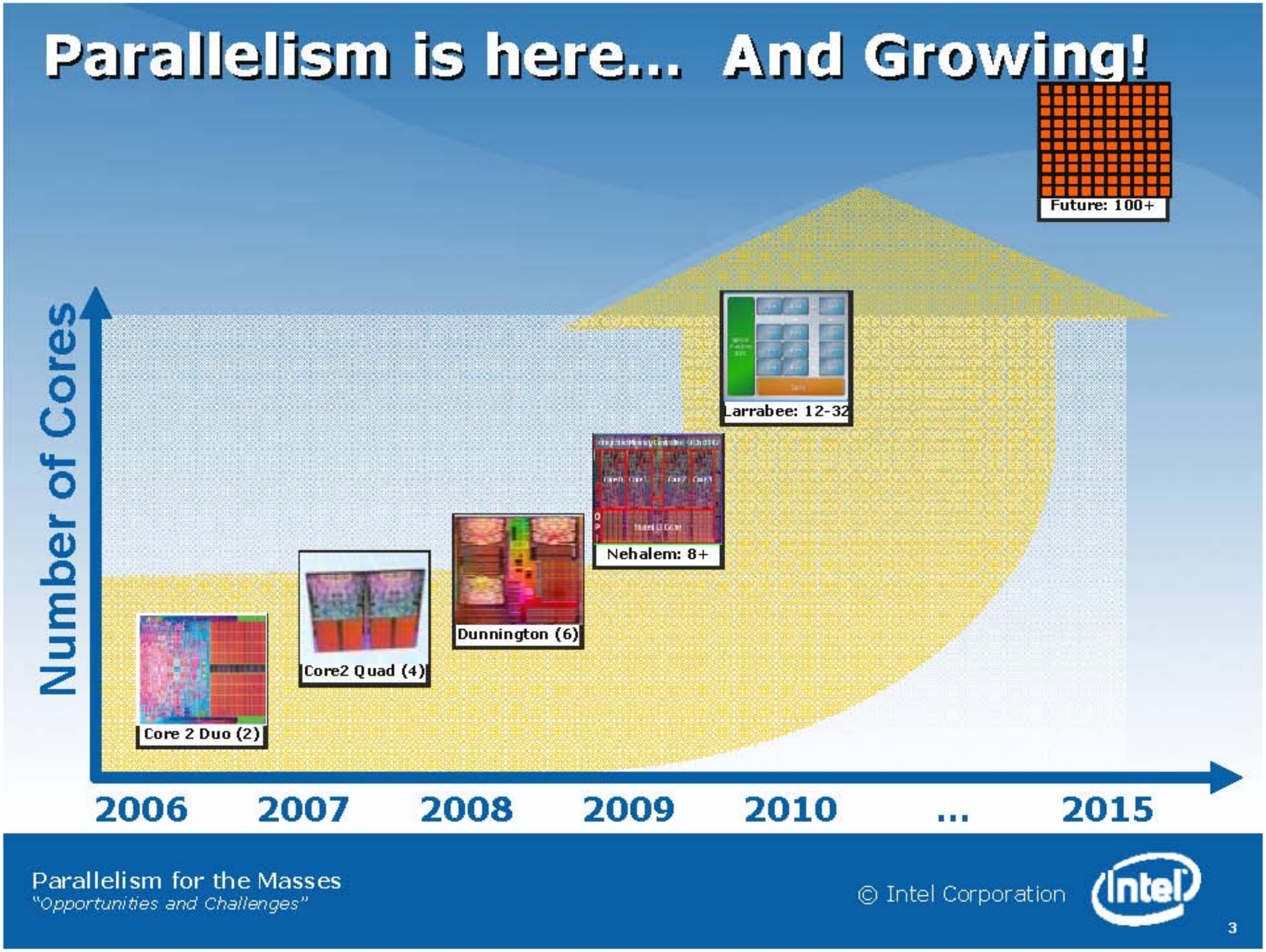
Parallel Thinking*

Guy Blelloch
Carnegie Mellon University

*PROBE as part of the Center for Computational Thinking



Andrew Chien, 2008

Parallel Thinking

How to deal with teaching parallelism?

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.

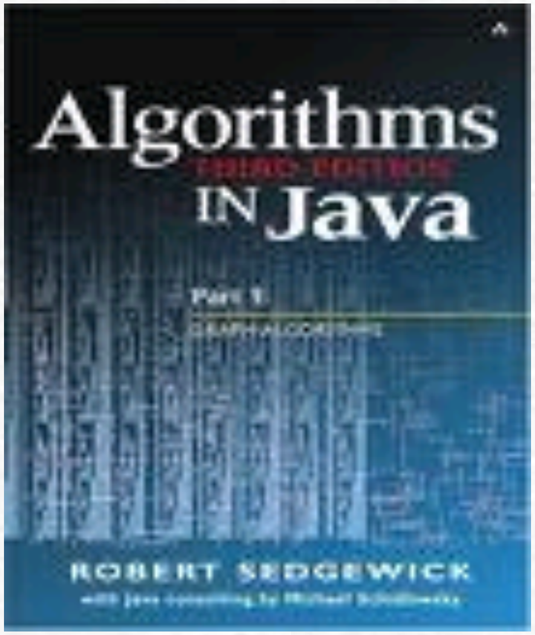
Option III : Teach parallelism from the start with sequential computing as a special case.

Parallel Thinking

- If explained at the right level of abstraction are many algorithms naturally parallel?
- If done right could parallel programming be as easy or easier than sequential programming for many uses?
- Are we currently **brainwashing** students to think sequentially?
- **What are the core parallel ideas that all computer scientists should know?**

Quicksort from Sedgewick

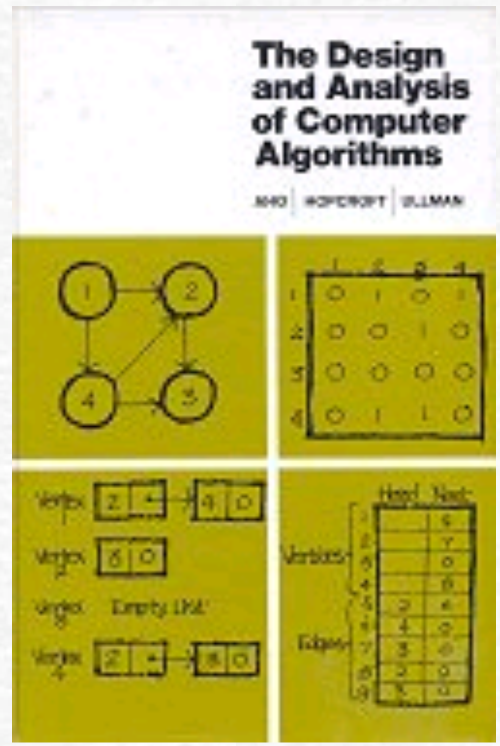
```
public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
      while (a[++i] < a[right]);
      while (a[right]<a[--j])
        if (j==left) break;
      if (i >= j) break;
      swap(a,i,j); }
    swap(a, i, right);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



Sequential!

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



Two forms of natural parallelism

Observation 1 and 2

- Natural parallelism is often lost in “low-level” implementations.
  - Need “higher level” descriptions
  - Need to revert back to the core ideas of an algorithm and recognize what is parallel and what is not
- Lost opportunity not to describe parallelism

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]

Each element gets sum of previous elements.
Seems sequential?

Scan

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

sum

[3, 6, 4, 12]

recurse

[0, 3, 9, 13]

sum

[2, 7, 12, 18]

interleave

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

Scan code

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

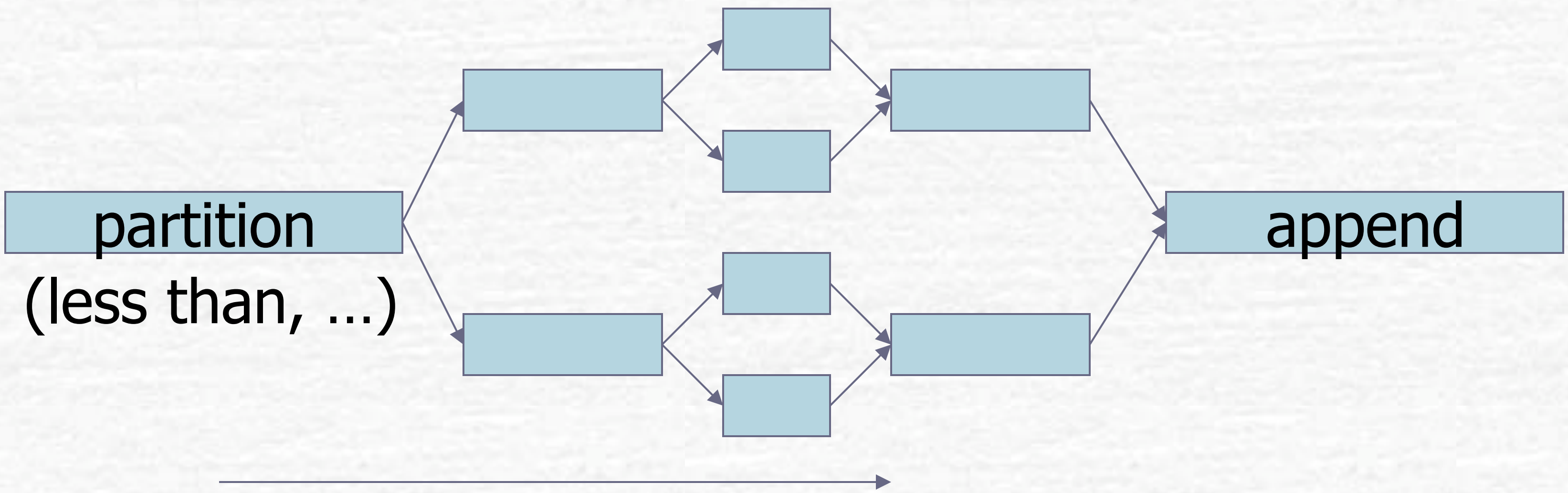
Observations 3, 4 and 5

- Just because it seems sequential does not mean it is
- + When in doubt recurse on a single smaller problem and use the result to solve larger problem
- + Transitions can be aggregated (composed)

+ Core parallel idea/technique

Qsort Complexity

Sequential Partition
Parallel calls



Span = O(n)

Work = O(n log n)

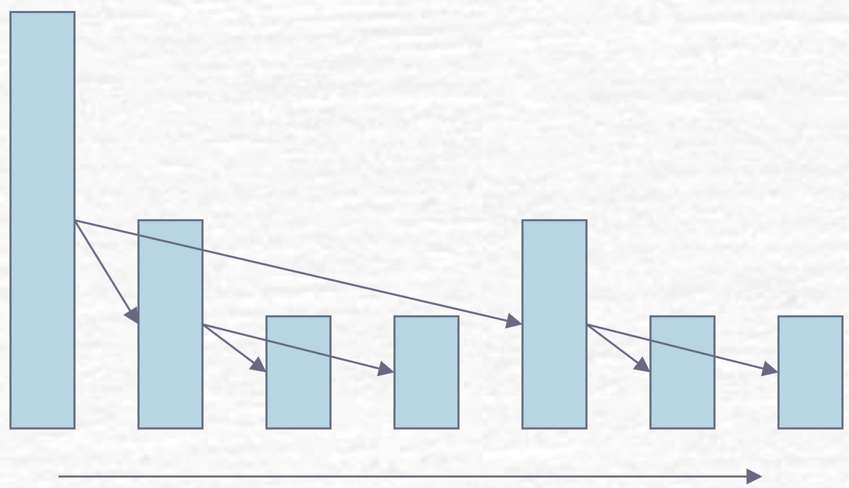
Not a very good parallel algorithm

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
```

Qsort Complexity

Parallel partition
Sequential calls



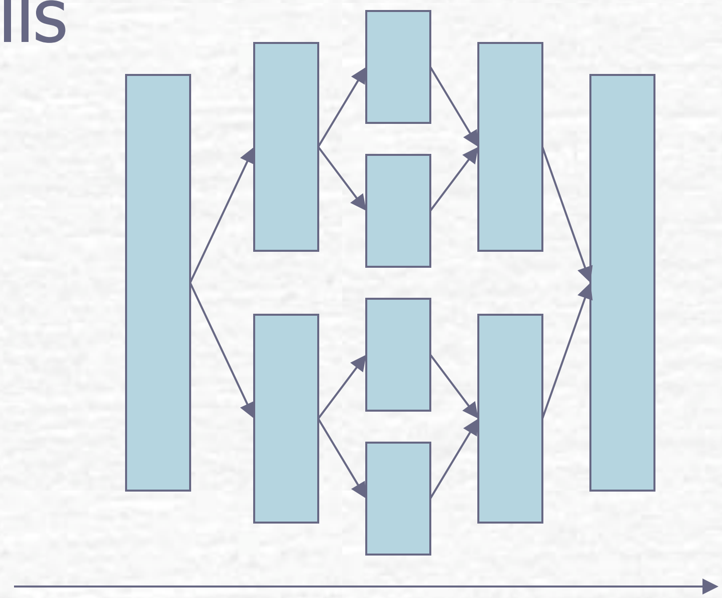
Span = O(n)

Work = O(n log n)

Still not a very good parallel algorithm

Qsort Complexity

Parallel partition
Parallel calls



Work = O(n log n)

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

A good parallel algorithm

Complexity in Nesl

Combining for parallel map:

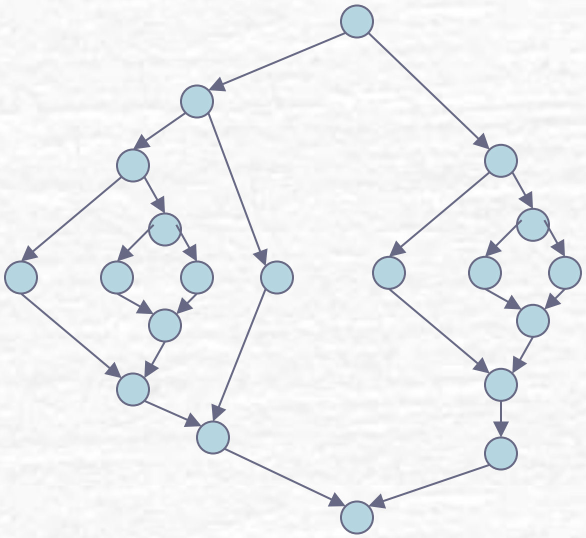
```
pexp = {exp(e) : e in A}
```

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

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

In general all you need is sum (work) and max (span) for nested parallel computations.

Generally for a DAG



- Any "greedy" schedule for a DAG with span (depth) D and work (size) W will complete in:
  $T < W/P + D$
- Any schedule will take at least:
  $T >= \max(W/P, D)$

Observations 6, 7, 8 and 9

- + Often need to take advantage of both “data parallelism” and “function parallelism”
- Abstract cost models that are not machine based are important.
- + Work and span are reasonable measures and can be easily composed with nested parallelism. No more difficult to understand than time in sequential algorithms.
- +′ Many ways to schedule

+′ = advanced topic

Matrix Inversion

```
Mat invert(mat M) {
  D⁻¹ = invert(D)
  S⁻¹ = A – BD⁻¹C
  S⁻¹ = invert(S)
  E = D⁻¹
  F = S⁻¹BD⁻¹
  G = -D⁻¹CS⁻¹
  H = D⁻¹ + D⁻¹CS⁻¹BD⁻¹
}
```

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$\begin{aligned} W(n) &= 2W(n/2) + 6W_*(n/2) \\ &= O(n^3) \end{aligned} \qquad \begin{aligned} D(n) &= 2D(n/2) + 6D_*(n/2) \\ &= O(n) \end{aligned}$$

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));
}
```

Quicksort in X10

```
double[] quicksort(double[] S) {
  if (S.length < 2) return S;
  double a = S[rand(S.length)];
  double[] S1,S2,S3;
  cnt = cnt+1;
  finish {
     async { S1 = quicksort(lessThan(S,a));}
     async { S2 = eqTo(S,a);}
     S3 = quicksort(grThan(S,a));
  }
  append(S1,append(S2,S3));
}
```

????

Observation 10

- Deterministic parallelism is important for easily understanding, analyzing and debugging programs.
  - Functional languages
  - Race detectors (e.g. cilkscreen)
  - Using non-functional languages in a functional style (is this safe?)

Atomic regions and transactions don't solve this problem.

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)
```

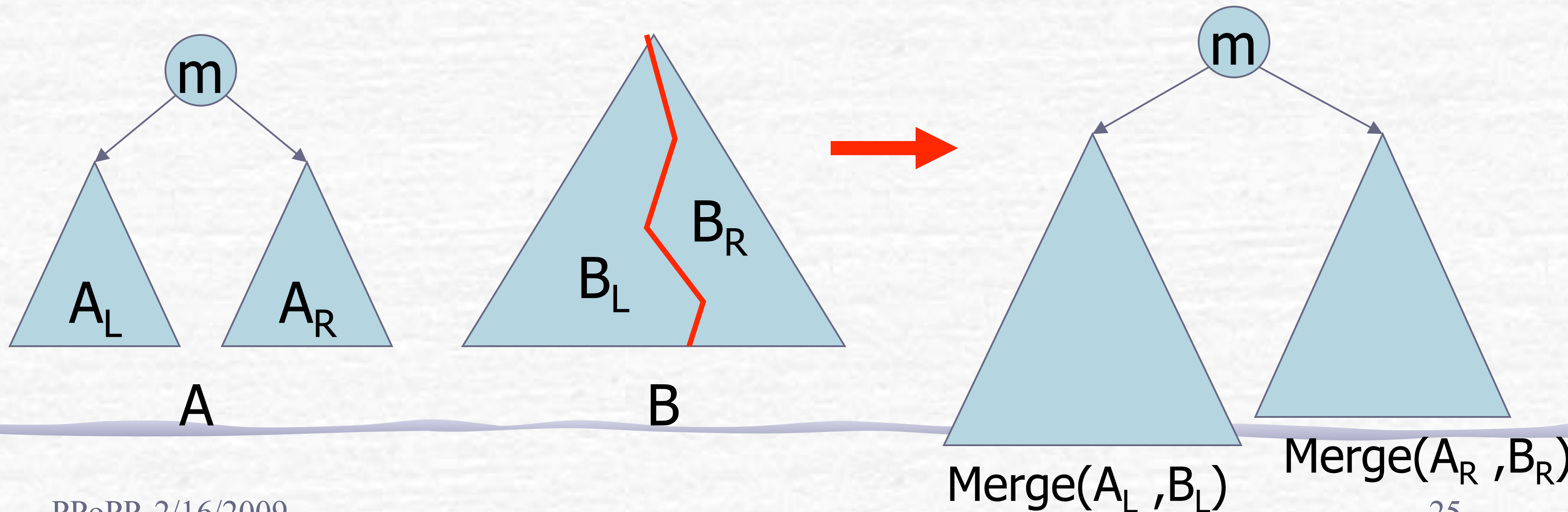
What about in parallel?

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 = $O(n)$

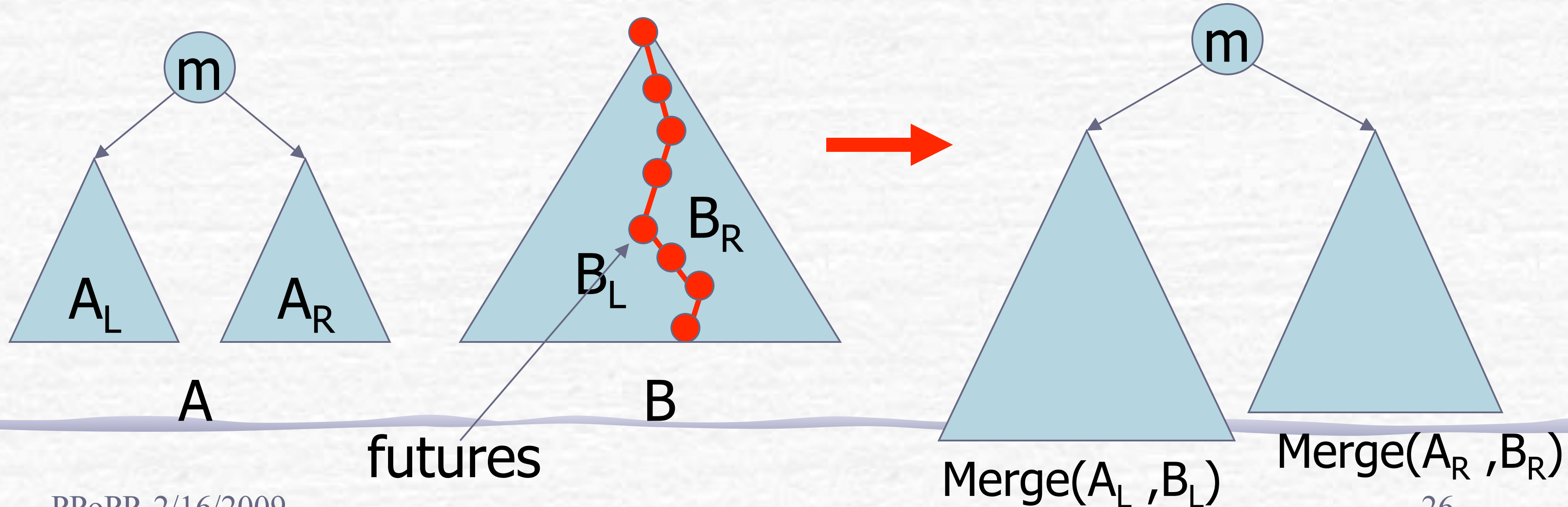
Merge in parallel



Merging with Futures

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

Span = O(log n)
Work = O(n)



Observations 11, 12 and 13

- + Divide and conquer even more useful in parallel than sequentially
- + Trees are better than lists for parallelism
- +′ Pipelining can asymptotically reduce depth, but can be hard to analyze

The Observations

General:

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

Model and Language:

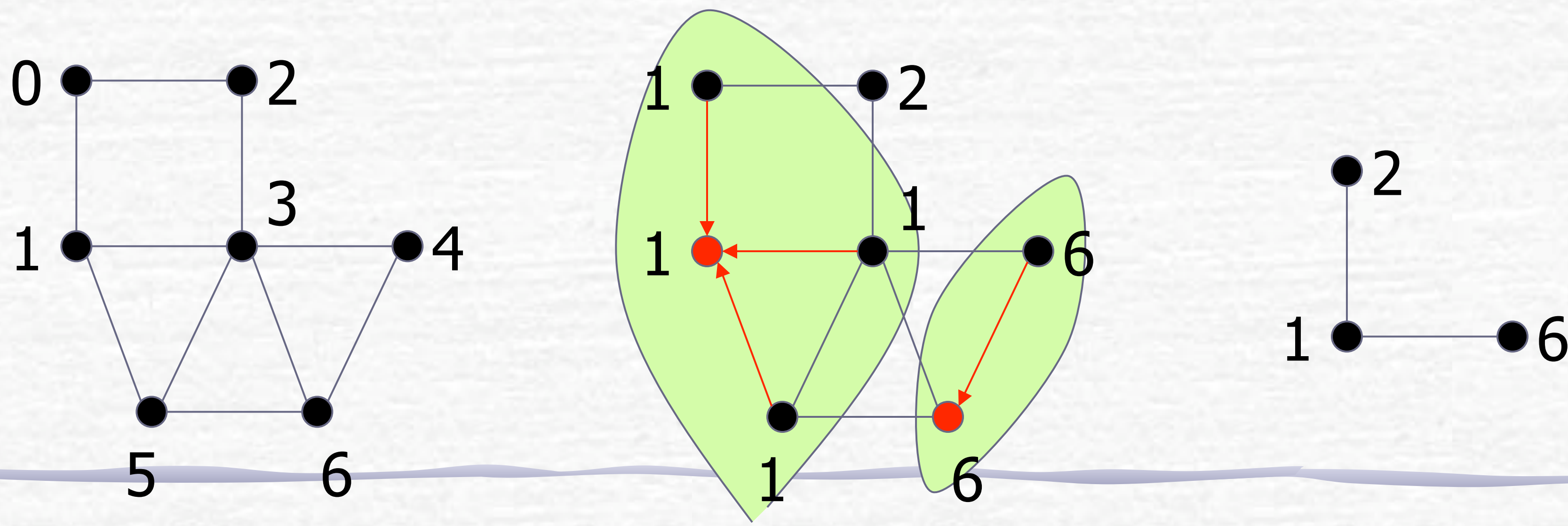
6. Need to take advantage of both “data” and “function” parallelism
7. Abstract cost models that are not machine based are important.
8. Work and span are reasonable measures
9. Many ways to schedule
10. Deterministic parallelism is important

Algorithmic Techniques

4. When in doubt recurse on a smaller problem
5. Transitions can be aggregated
11. Divide and conquer even more useful in parallel
12. Trees are better than lists for parallelism
13. Pipelining is useful, with care

More algorithmic techniques

- \+ Graph contraction
- \+ Identifying independent sets
- \+ Symmetry breaking
- \+ Pointer jumping



What else

Non-deterministic parallelism:
- Races and race detection
- Sequential consistency, serializability, linearizability, atomic primitives, locking techniques, transactions
- Concurrency models, e.g. the pi-calculus
- Lock and wait free algorithms

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

…

Excersise

- Identify the core ideas in Parallelism
  - Ideas that will still be useful in 20 years
  - Separate into “beginners” and “advanced”
- See how they fit into a curriculum
  - Emphasis on simplicity first
  - Will depend on existing curriculum

Possible course content

Biased by our current sequence

- 211: Fundamental data structures and algorithms
- 212: Principles of programming
- 213: Introduction to computer systems
- 251: Great theoretical ideals in computer science

211: Intro to Data Structures+Algos

Teach **deterministic nested parallelism** with **work and depth**.

- Introduce race conditions but don't allow them.
- **General techniques**: divide-and-conquer, contraction, combining, dynamic programming
- **Data structures**: stacks, queues, vectors, balanced trees, matrices, graphs,
- **Algorithms:** scan, sorting, merging, medians, hashing, fft, graph connectivity, MST

212: Principles of Programming

- Recursion, structural induction, currying
- Folding, mapping : emphasis on trees not lists
- Exceptions, parallel exceptions, and continuations
- Streams, futures, pipelining
- State and interaction with parallelism
- Nondeterminacy and linearizability
- Simple concurrent structure
- Or parallelism

213: Introduction to Systems

- Representing integers/floats
- Assembly language and atomic operations
- Out of order processing
- Caches, virtual memory, and memory consistency
- Threads and scheduling
- Concurrency, synchronization, transactions and serializability
- Network programming

Acknowledgements

This talk has been based on 30 years of research on parallelism by 100s of people.

Many ideas from the PRAM (theory) community and PL community

Conclusions/Questions

Should we teach parallelism from day 1 and sequential computing as a special case?

Could teaching parallelism actually make some things easier?

Are there a reasonably small number of core ideas that every undergraduate needs to know? If so, what are they?