CS 240A:

Parallel Prefix Algorithms

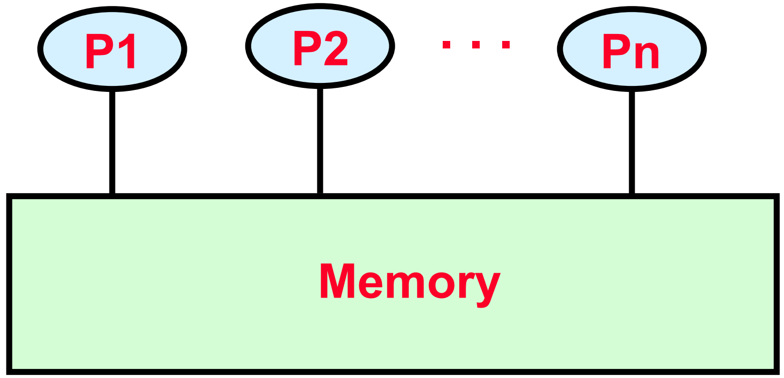
or

Tricks with Trees

Some slides from Jim Demmel,
Kathy Yelick, Alan Edelman,
and a cast of thousands …

PRAM model of parallel computation

Parallel
Random
Access
Machine



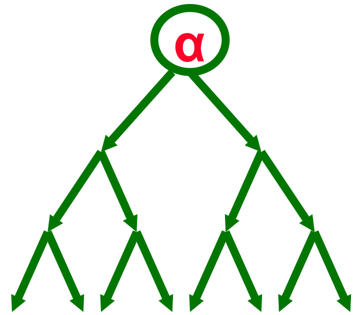
- Very simple theoretical model, used in 1970s and 1980s for lots of “paper designs” of parallel algorithms.
- Processors have unit-time access to any location in shared memory.
- Number of processors is allowed to grow with problem size.
- Goal is (usually) an algorithm with span $O(\log n)$ or $O(\log^2 n)$.
- Eg: Can you sort n numbers with $T_1 = O(n \log n)$ and $T_n = O(\log n)$?
  - Was a big open question until Cole solved it in 1988.
- Very unrealistic model but sometimes useful for thinking about a problem.

Parallel Vector Operations

- Vector add: $z = x + y$
  - Embarrassingly parallel if vectors are aligned; span = 1
- DAXPY: $v = \alpha*v + \beta*w$ (vectors v, w; scalar α, β)
  - Broadcast α & β, then pointwise vector +; span = log n
- DDOT: $\alpha = v^T*w$ (vectors v, w; scalar α)
  - Pointwise vector *, then sum reduction; span = log n

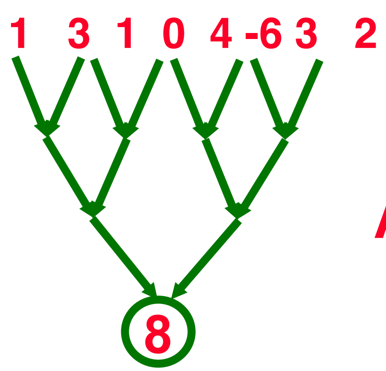
Broadcast and reduction

- Broadcast of 1 value to p processors with log p span



Broadcast

- Reduction of p values to 1 with log p span
- Uses associativity of +, *, min, max, etc.



Add-reduction

Parallel Prefix Algorithms

- A theoretical secret for turning serial into parallel
- Surprising parallel algorithms:

  If "there is no way to parallelize this algorithm!" …

- … it's probably a variation on parallel prefix!

Example of a prefix (also called a *scan*)

Sum Prefix

Input $x = (x_1, x_2, . . ., x_n)$

Output $y = (y_1, y_2, . . ., y_n)$

$$y_i = \Sigma_{j=1:i}\ x_j$$

Example

$x = (1, 2, 3, 4, 5, 6, 7, 8)$

$y = (1, 3, 6, 10, 15, 21, 28, 36)$

Prefix functions-- outputs depend upon an *initial* string

What do you think?

- Can we really parallelize this?

- It looks like this kind of code:

```
y(0) = 0;
for i = 1:n
    y(i) = y(i-1) + x(i);
```

- The ith iteration of the loop depends completely on the (i-1)st iteration.

- Work = n, span = n, parallelism = 1.

- Impossible to parallelize, right?

A clue?

x = (1, 2, 3, 4, 5, 6, 7, 8)

y = (1, 3, 6, 10, 15, 21, 28, 36)

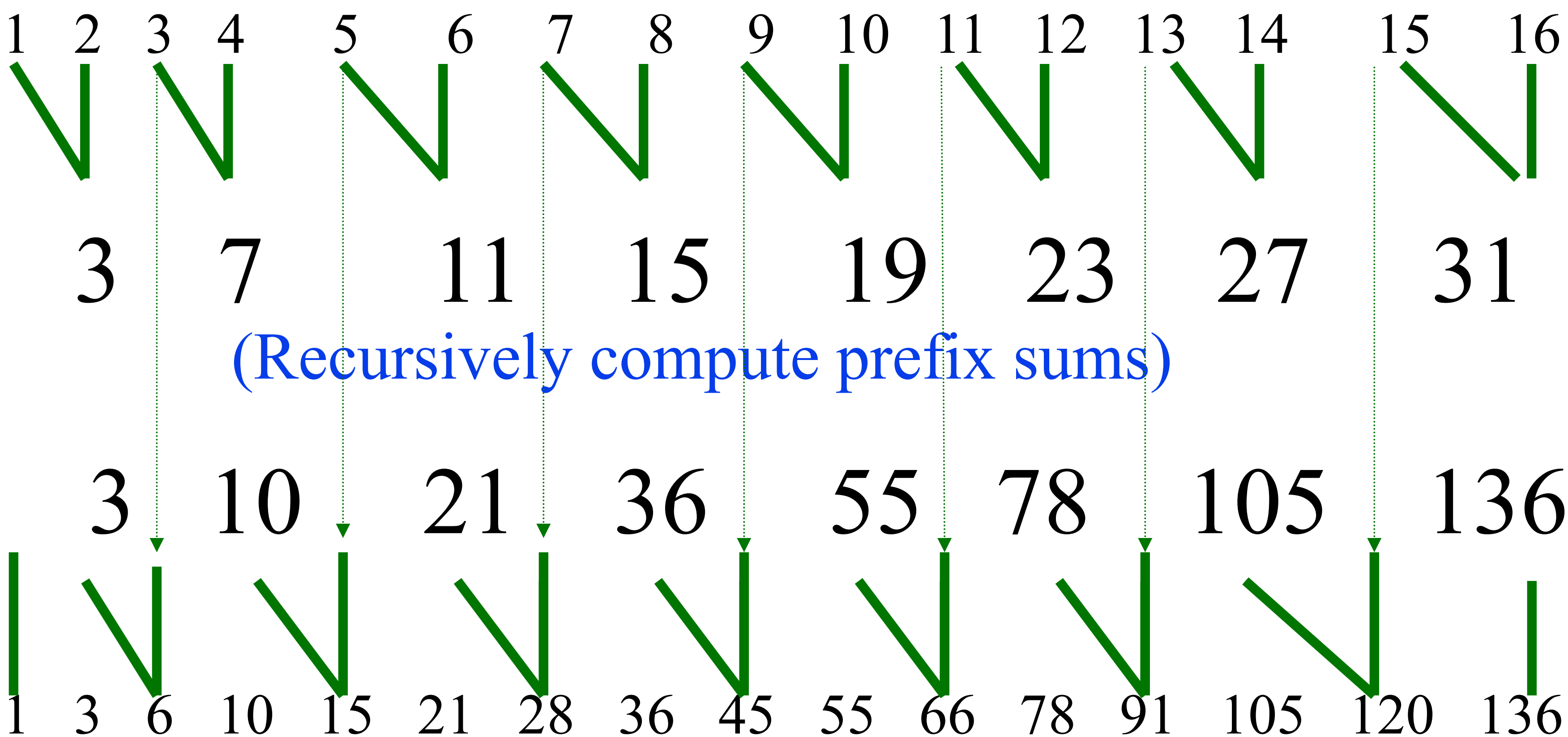
Is there any value in adding, say, 4+5+6+7?

If we separately have 1+2+3, what can we do?

Suppose we added 1+2, 3+4, etc. pairwise -- what could we do?

Prefix sum in parallel

Algorithm: **1. Pairwise sum** **2. Recursive prefix** **3. Pairwise sum**



Parallel prefix cost: Work and Span

- What's the total work?

1 2 3 4 5 6 7 8

Pairwise sums

3 7 11 15

Recursive prefix

3 10 21 36

Update "odds"

1 3 6 10 15 21 28 36

Parallel prefix cost: Work and Span

- What's the total work?

1 2 3 4 5 6 7 8

Pairwise sums

3 7 11 15

Recursive prefix

3 10 21 36

Update "odds"

1 3 6 10 15 21 28 36

- $T_1(n) = n/2 + n/2 + T_1(n/2) = n + T_1(n/2) = 2n - 1$

Parallel prefix cost: Work and Span

- What's the total work?

1 2 3 4 5 6 7 8

Pairwise sums

3 7 11 15

Recursive prefix

3 10 21 36

Update "odds"

1 3 6 10 15 21 28 36

- $T_1(n) = n/2 + n/2 + T_1(n/2) = n + T_1(n/2) = 2n - 1$
- $T_\infty(n) = 2 \log n$

Parallelism at the cost of twice the work!

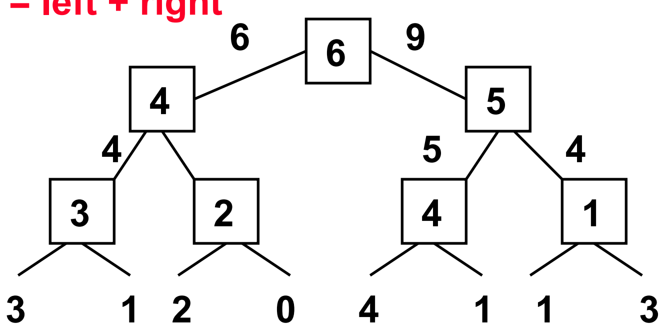
Non-recursive view of parallel prefix scan

- Tree summation: two phases
  - up sweep
    - get values L and R from left and right child
    - save L in local variable Mine
    - compute Tmp = L + R and pass to parent
  - down sweep
    - get value Tmp from parent
    - send Tmp to left child
    - send Tmp+Mine to right child

Up sweep:

mine = left

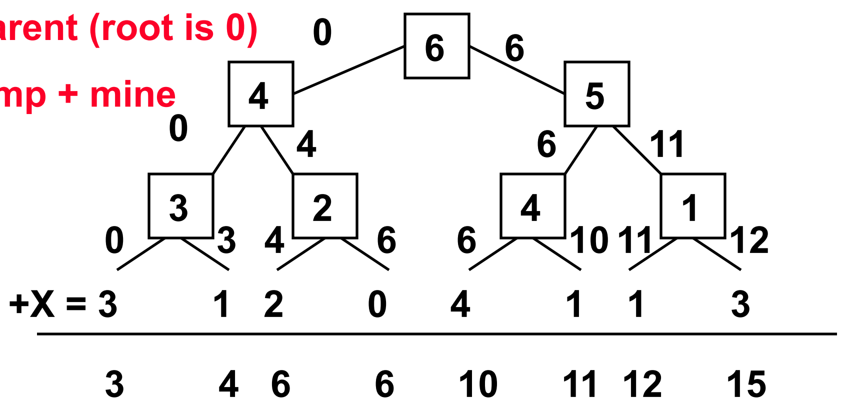
tmp = left + right



Down sweep:

tmp = parent (root is 0)

right = tmp + mine



Any associative operation works

Associative:

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

Sum (+) Product (*) Max Min
Input: Reals

All (and) Any (or)
Input: Bits (Booleans)

MatMul
Input: Matrices (not commutative!)

Scan (Parallel Prefix) Operations

- Definition: the parallel prefix operation takes a binary associative operator $\oplus$, and an array of n elements

$$[a_0, a_1, a_2, \ldots a_{n-1}]$$

and produces the array

$$[a_0, (a_0 \oplus a_1), \ldots (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

- Example: add scan of

[1, 2, 0, 4, 2, 1, 1, 3] is [1, 3, 3, 7, 9, 10, 11, 14]

Applications of scans

- Many applications, some more obvious than others

  - lexically compare strings of characters
  - add multi-precision numbers
  - add binary numbers fast in hardware
  - graph algorithms
  - evaluate polynomials
  - implement bucket sort, radix sort, and even quicksort
  - solve tridiagonal linear systems
  - solve recurrence relations
  - dynamically allocate processors
  - search for regular expression (grep)
  - image processing primitives

Using Scans for Array Compression

- Given an array of n elements

  $[a_0, a_1, a_2, \ldots a_{n-1}]$

  and an array of flags

  [1,0,1,1,0,0,1,…]

  compress the flagged elements into

  $[a_0, a_2, a_3, a_6, \ldots]$

- Compute an add scan of [0, flags] :

  [0,1,1,2,3,3,4,…]

- Gives the index of the i^{th} element in the compressed array
  - If the flag for this element is 1, write it into the result array at the given position

Array compression: Keep only positives

Matlab code

```
% Start with a vector of n random #s
% normally distributed around 0.

A = randn(1,n);
flag = (A > 0);
addscan = cumsum(flag);
parfor i = 1:n
    if flag(i)
        B(addscan(i)) = A(i);
    end;
end;
```

Fibonacci via Matrix Multiply Prefix

$$\boxed{F_{n+1} = F_n + F_{n-1}}$$

$$\boxed{\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}}$$

Can compute all F_n by matmul_prefix on

$$\left[\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \right]$$

then select the upper left entry

Carry-Look Ahead Addition (Babbage 1800's)

Example	
1 0 1 1 1	Carry
1 0 1 1 1	First Int
1 0 1 0 1	Second Int
1 0 1 1 0 0	Sum

Goal: Add Two n-bit Integers

Carry-Look Ahead Addition (Babbage 1800's)

Goal: Add Two n-bit Integers

Example		Notation
1 0 1 1 1	Carry	c_2 c_1 c_0
1 0 1 1 1	First Int	a_3 a_2 a_1 a_0
1 0 1 0 1	Second Int	b_3 b_2 b_1 b_0

Carry-Look Ahead Addition (Babbage 1800's)

Goal: Add Two n-bit Integers

Example						Notation			
1	0	1	1	1		Carry	c_2	c_1	c_0	
	1	0	1	1	1	First Int	a_3	a_2	a_1	a_0
	1	0	1	0	1	Second Int	b_3	b_2	b_1	b_0

(addition mod 2)

$c_{-1} = 0$

for $i = 0 : n-1$

$\quad s_i = a_i + b_i + c_{i-1}$

$\quad c_i = a_i b_i + c_{i-1}(a_i + b_i)$

end

$s_n = c_{n-1}$

Goal: Add Two n-bit Integers

Example							Notation			
1	0	1	1	1		Carry	c_2	c_1	c_0	
	1	0	1	1	1	First Int	a_3	a_2	a_1	a_0
	1	0	1	0	1	Second Int	b_3	b_2	b_1	b_0

$c_{-1} = 0$

for $i = 0 : n-1$

$\quad s_i = a_i + b_i + c_{i-1}$

$\quad c_i = a_i b_i + c_{i-1}(a_i + b_i)$

end

$s_n = c_{n-1}$

$$\begin{bmatrix} c_i \\ 1 \end{bmatrix} = \begin{bmatrix} a_i + b_i & a_i b_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{i-1} \\ 1 \end{bmatrix}$$

(addition mod 2)

Goal: Add Two n-bit Integers

Example							Notation			
1	0	1	1	1		Carry	c_2	c_1	c_0	
	1	0	1	1	1	First Int	a_3	a_2	a_1	a_0
	1	0	1	0	1	Second Int	b_3	b_2	b_1	b_0

$c_{-1} = 0$

for $i = 0 : n-1$

$\quad s_i = a_i + b_i + c_{i-1}$

$\quad c_i = a_i b_i + c_{i-1}(a_i + b_i)$

end

$s_n = c_{n-1}$

$$\begin{bmatrix} c_i \\ 1 \end{bmatrix} = \begin{bmatrix} a_i + b_i & a_i b_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{i-1} \\ 1 \end{bmatrix}$$

1. compute c_i by binary matmul prefix
2. compute $s_i = a_i + b_i + c_{i-1}$ in parallel

Adding two n-bit integers in O(log n) time

- Let a = a[n-1]a[n-2]…a[0] and b = b[n-1]b[n-2]…b[0] be two n-bit binary numbers
- We want their sum s = a+b = s[n]s[n-1]…s[0]

```
c[-1] = 0          … rightmost carry bit
for i = 0 to n-1
    c[i] = ( (a[i] xor b[i])  and  c[i-1] )  or  ( a[i]  and  b[i] )   ... next carry bit
    s[i] = a[i] xor b[i] xor c[i-1]
```

- Challenge: compute all c[i] in O(log n) time via parallel prefix

for all (0 <= i <= n-1) p[i] = a[i] xor b[i] … propagate bit
for all (0 <= i <= n-1) g[i] = a[i] and b[i] … generate bit

$$\begin{bmatrix} c[i] \\ 1 \end{bmatrix} = \begin{bmatrix} (\ p[i] \text{ and } c[i-1]\ ) \text{ or } g[i] \\ 1 \end{bmatrix} = \begin{bmatrix} p[i] & g[i] \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} c[i-1] \\ 1 \end{bmatrix} = M[i] * \begin{bmatrix} c[i-1] \\ 1 \end{bmatrix}$$

… 2-by-2 Boolean matrix multiplication (associative)

$$= M[i] * M[i-1] * \ldots M[0] * \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

… evaluate each product M[i] * M[i-1] * … * M[0] by parallel prefix

- Used in all computers to implement addition - Carry look-ahead

Segmented Operations

Inputs = ordered pairs (operand, boolean)

e.g. (x, T) **or** (x, F)

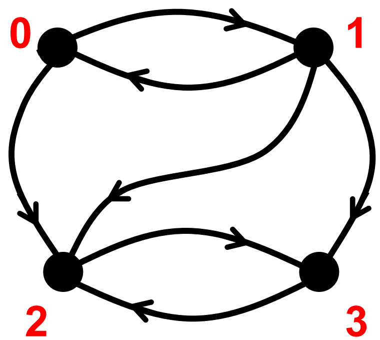
Change of segment indicated by switching T/F

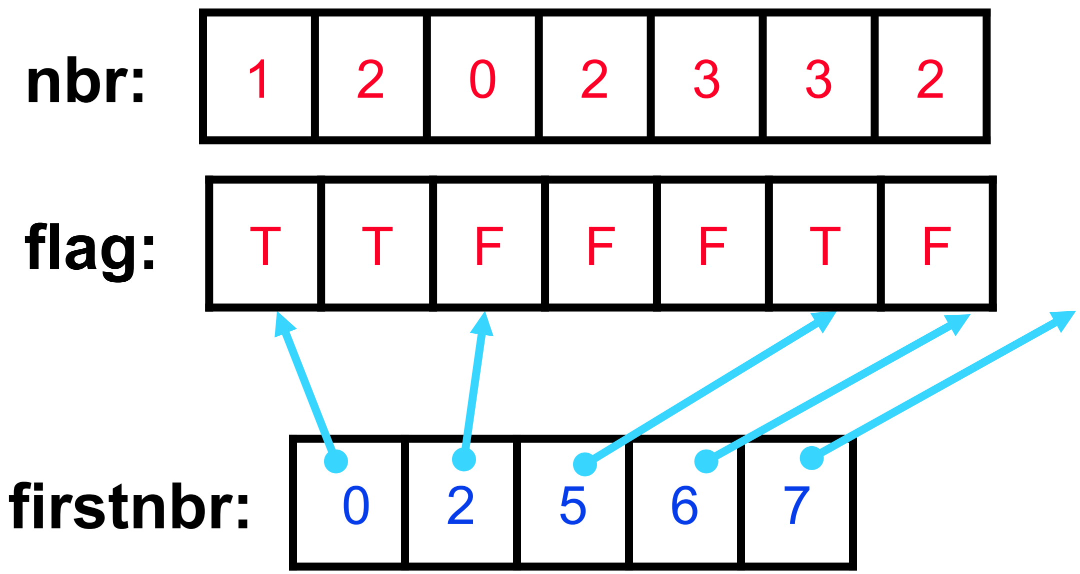
$\oplus_2$	(y, T)	(y, F)
(x, T)	($x \oplus y$, T)	(y, F)
(x, F)	(y, T)	($x \oplus y$, F)

e. g.	1	2	3	4	5	6	7	8
	T	T	F	F	F	T	F	T
Result	1	3	3	7	12	6	7	8

Any Prefix Operation May Be Segmented!

Graph algorithms by segmented scans





The usual CSR data structure, plus segment flags!

Multiplying n-by-n matrices in O(log n) span

- For all (1 <= i,j,k <= n) P(i,j,k) = A(i,k) * B(k,j)
  - span = 1, work = n^3

- For all (1 <= i,j <= n) C(i,j) = Σ P(i,j,k)
  - span = O(log n), work = n^3 using a tree

Inverting dense n-by-n matrices in $O(\log^2 n)$ span

- Lemma 1: Cayley-Hamilton Theorem
  - expression for A^{-1} via characteristic polynomial in A
- Lemma 2: Newton's Identities
  - Triangular system of equations for coefficients of characteristic polynomial
- Lemma 3: $\mathrm{trace}(A^k) = \sum_{i=1}^{n} A^k[i,i] = \sum_{i=1}^{n} [\lambda_i(A)]^k$
- Csanky's Algorithm (1976)

1) Compute the powers $A^2, A^3, \ldots, A^{n-1}$ by parallel prefix
span = $O(\log^2 n)$
2) Compute the traces $s_k = \mathrm{trace}(A^k)$
span = $O(\log n)$
3) Solve Newton identities for coefficients of characteristic polynomial
span = $O(\log^2 n)$
4) Evaluate A^{-1} using Cayley-Hamilton Theorem
span = $O(\log n)$

- Completely numerically unstable

Evaluating arbitrary expressions

- Let E be an arbitrary expression formed from +, -, *, /, parentheses, and n variables, where each appearance of each variable is counted separately
- Can think of E as arbitrary expression tree with n leaves (the variables) and internal nodes labelled by +, -, * and /
- Theorem (Brent): E can be evaluated with O(log n) span, if we reorganize it using laws of commutativity, associativity and distributivity
- Sketch of (modern) proof: evaluate expression tree E greedily by
    - collapsing all leaves into their parents at each time step
    - evaluating all “chains” in E with parallel prefix

The myth of log n

- **The $\log_2$ n span is not the main reason for the usefulness of parallel prefix.**

- **Say n = 1000000p** (1000000 summands per processor)
    - Cost = $\boxed{\text{(2000000 adds)}}$ + ($\log_2 P$ message passings)

fast & embarassingly parallel

(2000000 local adds are serial for each processor, of course)

Summary of tree algorithms

- Lots of problems can be done quickly - in theory - using trees
- Some algorithms are widely used
  - broadcasts, reductions, parallel prefix
  - carry look ahead addition
- Some are of theoretical interest only
  - Csanky's method for matrix inversion
  - Solving tridiagonal linear systems (without pivoting)
  - Both numerically unstable
  - Csanky does too much work
- Embedded in various systems
  - CM-5 hardware control network
  - MPI, UPC, Titanium, NESL, other languages