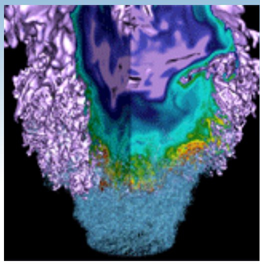
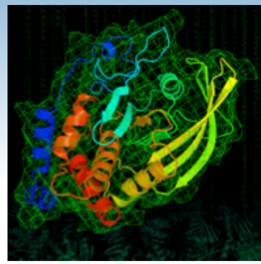
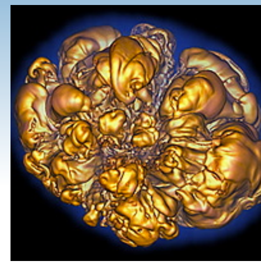
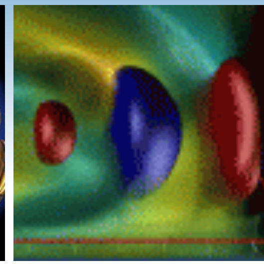
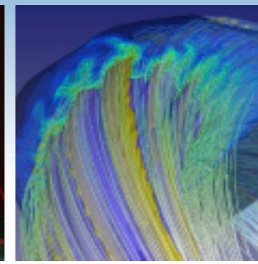
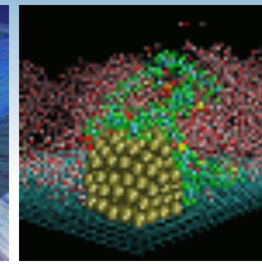
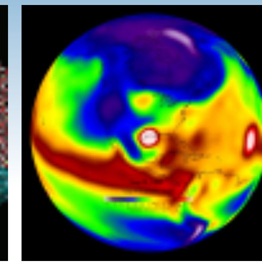








Modern Parallel Languages

Kathy Yelick

Lecture 2: Data parallelism (part 1)
NESL

http://www.eecs.berkeley.edu/~yelick/cs294-f13

Data parallelism

- No widely-accepted *clear* definition
- Wikipedia: "data parallelism is typically expressed as a single thread of control operating on data sets *distributed over all nodes*"
- Wikipedia: "But it is said that a data parallel language has a notion of explicit parallelism too"
- *Ask:* Data parallelism focuses on distributing the data across different parallel computing nodes. It contrasts with task parallelism.
- *Microsoft:* Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array.

Data parallel algorithms / models

- Hillis and Steele

general communications. We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control. The intent is not so much to present new

- Blelloch
- ..data-parallel models, the *parallel vector models*. The definition is based on a machine that can store a vector in each memory location and whose instructions operate on these vectors as a whole—for example, elementwise adding two equal length vectors. In the model, each vector instruction requires one “program step”.

Our definition for this class

- A (pure) data parallel language has
  - A single thread of control, i.e., a serial semantics, which means all behaviors we can see in parallel can also be observed in the serial execution
  - It has operations on aggregate data structures (collections) to (implicitly) express parallelism
- These have a limited expressiveness, but clean and intuitive semantics
- Collections-oriented languages exist independent of parallelism

Collection-Oriented Languages

- **Languages that support actions on large collections of data with a single operation**
- **Examples:**
  - **FORTRAN 90 and arrays**
  - **APL and arrays,**
  - **Connection Machine LISP and xectors**
  - **PARALATION LISP and paralations**
  - **SETL and sets**
  - **Haskell / Miranda features, i.e., comprehensions**
- **Many of these were developed before parallelism became "important" (i.e., pre-1980s)**

Sipelstein, Jay M. and Blelloch, Guy E., "Collection-oriented languages" (1990). Computer Science Department. Paper 2006. http://repository.cmu.edu/compsci/2006

Features in Collection-Oriented Languages

- Unary Apply-to-each, e.g., negate elements of vector A
  - Implicit: -A (APL)
  - Explicit: α- [3,1,4] (CM Lisp) or {-e : e in A } (SETL)
- Non-unary Apply-to-each
  - E.g., implicit A+B
  - Element correspondence: which elements line up?
  - Element extension: adding a scalar to a vector
- Rearranging elements
  - E.g., Permute according to a list of indices (source or target)
- Nesting: can collections contain collections?
- Homogeneity: are all elements of the same type?

Tradeoffs?

Sipelstein, Jay M. and Blelloch, Guy E., "Collection-oriented languages" (1990). Computer Science Department. Paper 2006. http://repository.cmu.edu/compsci/2006

Examples of collection-oriented languages

FP	
`(/+)∘(αx)∘trans`	A = [1 0 5 3] B = [3 4 3 7] ⇒ 3 + 0 + 15 + 21 = 39

Compute the dot product of two vectors

APL	
`+/(A*(*\1,(((ρA)−1)ρx)))`	A = [1 2 3 4] x = 2 ⇒ $1 + 2 \times 2 + 3 \times 2^2 + 4 \times 2^3 = 41$

Evaluate a polynomial with given coefficients A at value x

CM-LISP

```
(let ((l (length A))
      (p (α (β+ A →1.0) αl)))
   (- (β+ (α* p (αlg p)))))
```

A = [a b c a d c b d]
⇒ 2

Compute Shannon entropy of A: $H(i) = -\sum p(i) \lg p(i)$
where $p(i)$ is the probability that i occurs in the input string, for each i.

Sipelstein, Jay M. and Blelloch, Guy E., "Collection-oriented languages" (1990). Computer Science Department. Paper 2006. http://repository.cmu.edu/compsci/2006

More examples

SETL	

<pre>a := [2..n];
result := {};
loop while #a > 0 do
    p := first a;
    a := [x in a | (x mod p) /= 0];
    result := result with p;
end;
print result;</pre>
	N = 10
	⇒ [2 3 5 7]

Find prime numbers with the Sieve of Erastosthenes

FORTRAN 90	

<pre>R[2:n-1] =
  (F[1:n-2]-2*F[2:n-1]+F[3:n])/(d*d)</pre>
	F = [1 2 2 3 4]
	R[2:4] ⇒ [-.1 .1 0]

Compute the second derivative of F given a vector of values

Sipelstein, Jay M. and Blelloch, Guy E., "Collection-oriented languages" (1990). Computer Science Department. Paper 2006. http://repository.cmu.edu/compsci/2006

NESL Goals

- Data-parallelism (based on sequences):
  - Apply functions to sequence
  - Operate on sequence (e.g., permute)
- To support complete nested parallelism
  - Nested sequences
  - Applying user-defined functions on sequences, including parallel functions
- Efficient code for SIMD and MIMD machines
- Good for describing parallel algorithms
  - Each function has two complexity measures: work and depth, which can be mapped to a VRAM model

Readability (no races)

Expressive-ness (generality)

Performance & portability

Performance transparency

NESL Overview

- Strongly typed
- Functional
- Strict (vs. Lazy)
  - E.g., what does this statement do?

    print length([2+1, 3*2, 1/0, 5-4])
  - Is this just an implementation issue?
  - Why do we care?
- Nested Data-parallel

Readability
Safety
Performance?

Readability (modularity)

Performance?

Claim: NESL is for “Hard” Parallel 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!

Outline

A partial list of algorithms that use scans

- A log n lower bound to compute any function in parallel
- Reduction and broadcast in O(log n) time
- Parallel prefix (scan) in O(log n) time
- Adding two n-bit integers in O(log n) time
- Multiplying n-by-n matrices in O(log n) time
- Inverting n-by-n triangular matrices in $O(\log^2 n)$ time
- Inverting n-by-n dense matrices in $O(\log^2 n)$ time
- Evaluating arbitrary expressions in O(log n) time
- Evaluating recurrences in O(log n) time
- "2D parallel prefix", for image segmentation (Catanzaro & Keutzer)
- Sparse-Matrix-Vector-Multiply (SpMV) using Segmented Scan
- Parallel page layout in a browser (Leo Meyerovich, Ras Bodik)
- Solving n-by-n tridiagonal matrices in O(log n) time
- Traversing linked lists
- Computing minimal spanning trees
- Computing convex hulls of point sets…

Tricks with Trees

(revisited from CS267)

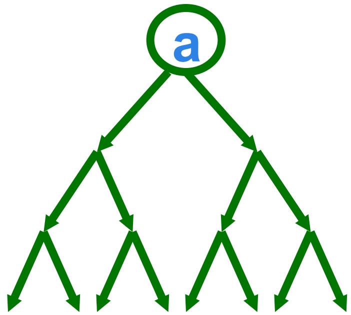
Some slides from John Gilbert, who borrowed some from Jim Demmel, Kathy Yelick ☺, Alan Edelman, and a cast of thousands …

Parallel Vector Operations

- Vector add: $z = x + y$
  - Embarrassingly parallel if vectors are aligned
- DAXPY: $z = a*x + y$ (a is scalar)
  - Broadcast a, followed by independent * and +
- DDOT: $s = x^T y = \Sigma_j x[j] * y[j]$
  - Independent * followed by + reduction

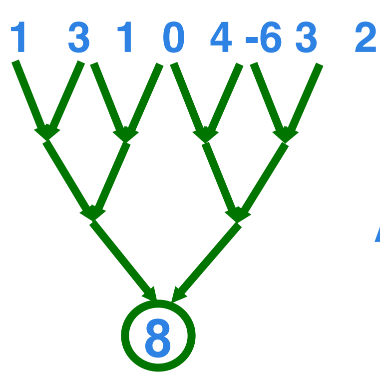
Broadcast and reduction

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



Broadcast

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



Add-reduction

Example of a prefix

<u>Sum Prefix</u>

Input $x = (x1, x2, \ldots, xn)$

Output $y = (y1, y2, \ldots, yn)$

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

<u>Example</u>

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

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

<u>Prefix</u> 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.
- Impossible to parallelize, right?

A clue?

x = (1, 2, 3, 4, 5, 6, 7, 8)
y = (1, 3, 6, 10, 15, 21, 28, 36)

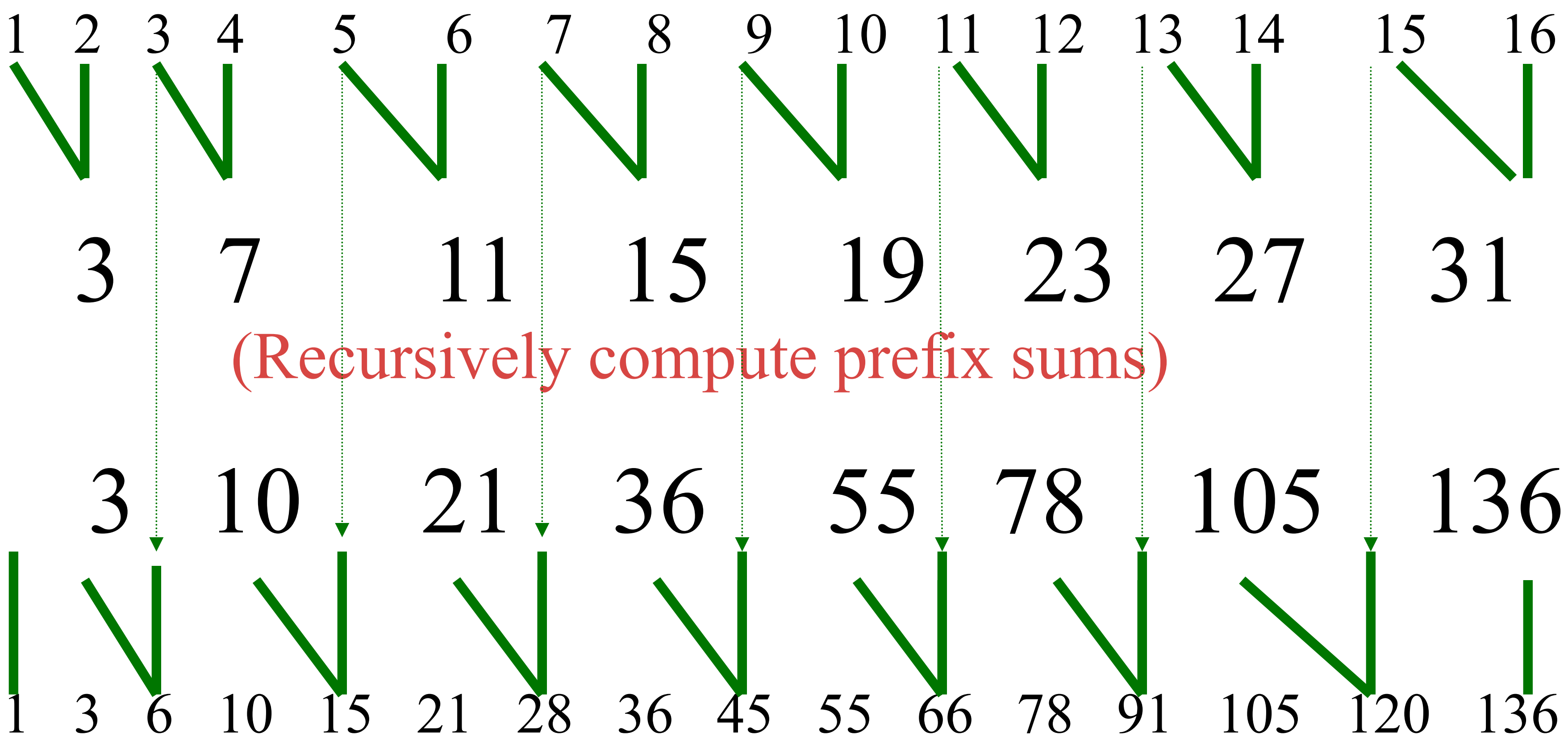
Is there any value in adding, say, 4+5+6+7?

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

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

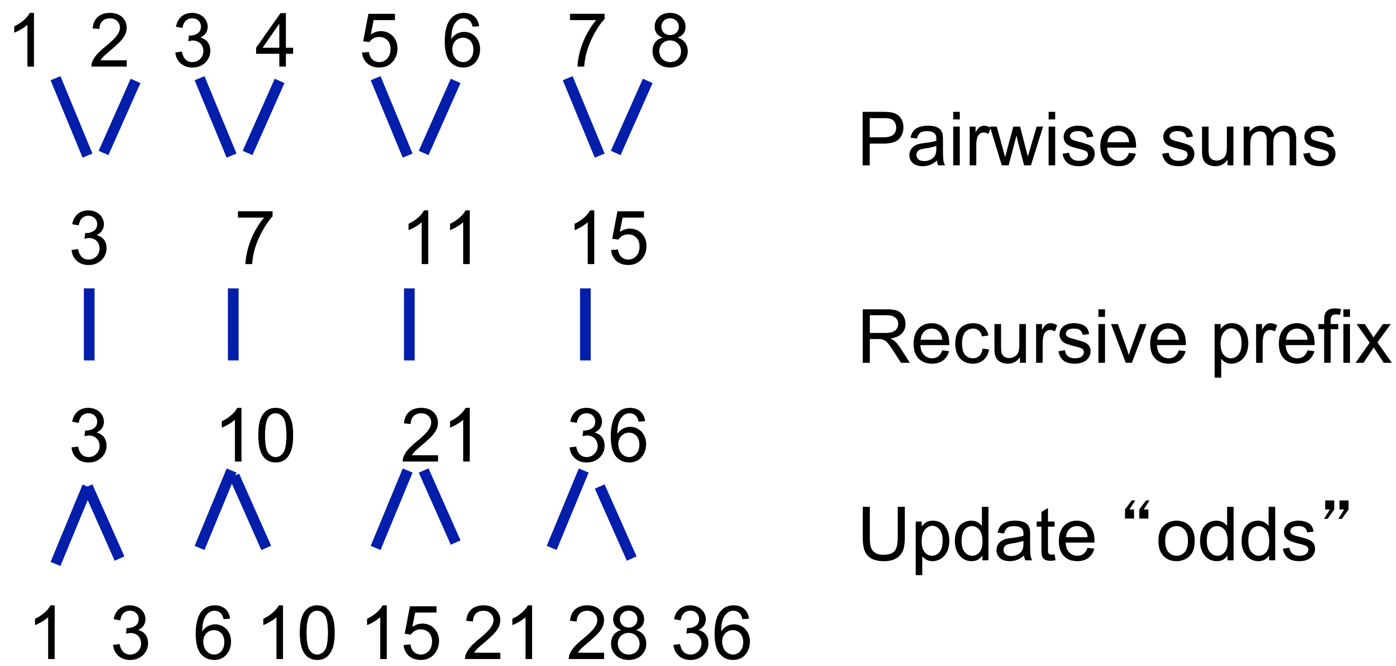
Prefix sum in parallel

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



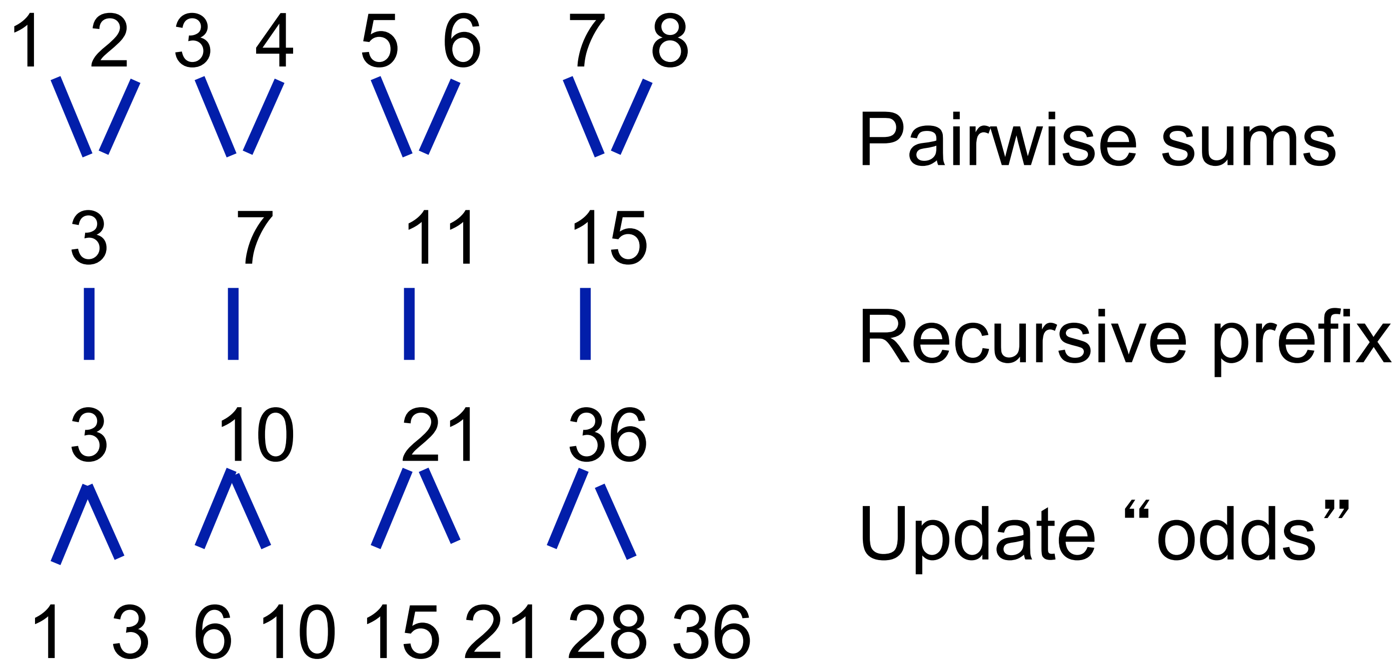
Parallel prefix cost

- What's the total work?



Parallel prefix cost

- What's the total work?



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

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 more work (2x)

Historical: Hillis and Steele algorithm does n reductions

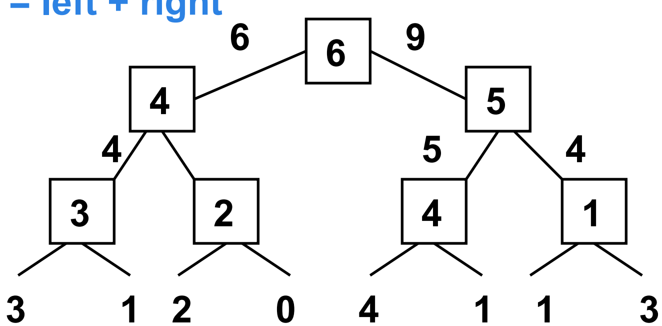
Non-recursive view of parallel prefix scan

- 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

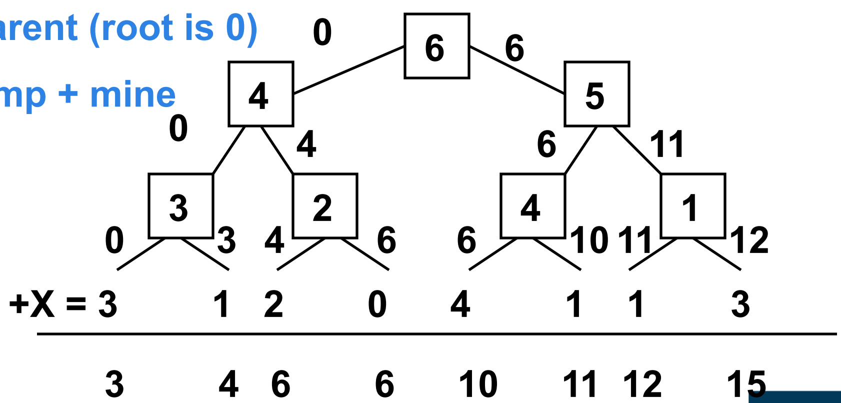
tmp = left + right



Down sweep:

tmp = parent (root is 0)

right = tmp + mine



Blelloch algorithm (?)

Scan (Parallel Prefix) Operations

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

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

and produces the array

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

- Example: add scan of

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

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 (Boolean)

MatMul

Input: Matrices

Lexical analysis

Input: Strings

Lexical analysis (tokenizing, scanning)

- Given a language of:
  - Identifiers: string of chars
  - Strings: in double quotes
  - Ops: +,-,*,=,<,>,<=, >=

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

- Lexical analysis
  - Replace every character in the string with the array representation of its state-to-state function (column).
  - Perform a parallel-prefix operation with ⊕ as the array composition. Each character becomes an array representing the state-to-state function for that prefix.
  - Use initial state (row 1) to index into these arrays.

Hillis and Steele, CACM 1986

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

E.g., 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

Segmented Operations

Inputs = Ordered Pairs
(operand, boolean)
e.g. (x, T) or (x, F)

Change of segment indicated by switching T/F

$+_2$	(y, T)	(y, F)
(x, T)	(x+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

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 = (2000000 adds) + ($\log_2 P$ message passings)

fast & embarassingly parallel

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

Key to implementing NESL Efficiently on Clusters, MPPs (aka MIMD machines)

VRAM Model: Vector Random-Access Machine

- VRAM from Blelloch, similar to PRAM
- Assumes scan operations can be done in O(1) time

- On a PRAM, a scan takes O(log n) time, so could apply an O(log n) factor to get PRAM complexity
- Assumption: organizing based on vectors makes complexity analysis easier, examples of performance
  - # Vector (length) O(1)
  - Sum(Vector) O(1)
  - Permute (Vector, Index Vector) O(1)
  - Add O(1)
  - Scan (Vector) O(1)
  - Max (Vector) O(1)

NESL : In a nutshell

Simple Call-by-Value Functional Language
- + Built in Parallel type (nested sequences)
- + Parallel map (apply-to-each)
- + Parallel aggregate operations
- + Cost semantics (work and depth)

Sequential Semantics

Some non-pure features at "top level"

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

NESL : History

- Developed in 1990
- Implemented on CM, Cray, MPI, and sequentially using a stack based intermediate language
- Interactive environment with remote calls
- Over 100 algorithms and applications written – used to teach parallel algorithms
- Mostly dormant since 1997

Slide: Blelloch “NESL Revisited”, Intel Workshop 2006

NESL: Parallel Operations on Sequences

- Sequences:
  - [1, 2, 9, -3]
  - {negate(a) : a in [2, -4, -9, 5]} → [-2, 4, 9, -5]
- No restrictions on functions that can be applied
  - Why does this work?
- Nested parallelism
  - flatten ([[2, 1], [7, 3, 0], [4]]) → [2, 1, 7, 3, 0, 4]

NESL: Parallel Map

```
A = [3.0, 1.0, 2.0]
B = [[4, 5, 1, 6], [2], [8, 11, 3]]
C = "Yoknapatawpah County"
D = ["the", "rain", "in", "Spain"]
```

Sequence Comprehensions:

```
{x + .5 : x in A} -> [3.5, 1.5, 2.5]
{sum(b) : b in B} -> [16, 2, 22]
{c in C | c < 'n} -> "kaaaahc"
{w[0] : w in D}   -> "triS"
```

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

NESL : Aggregate Operations

```
A = [3.0, 1.0, 2.0]
D = ["the", "rain", "in", "Spain"]
E = [(3,"Italy"), (1,"sun")]
```

Parallel write : `['a] * [int*'a] -> ['a]`

```
D <- E    -> ["the","sun","in","Italy"]
```

Prefix sum : `('a*'a->'a)*'a*['a] -> ['a]*'a`

```
scan('+,2.0,A) -> ([2.0,5.0,6.0],8.0)
  plus_scan(A)    -> [0.0,3.0,4.0]
  sum(A)          -> 6.0
```

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

NESL: Cost Model

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 PRAM:

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

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

Example : Quicksort (Version 1)

```
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};
in quicksort(S1) ++ S2 ++ quicksort(S3);
```

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

Example : Quicksort

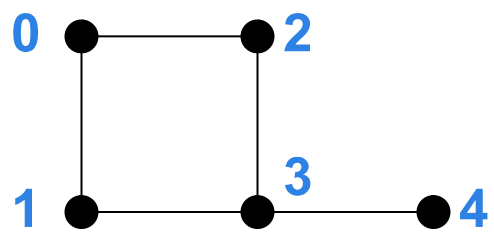
```
function quicksort(S) =
if (#S <= 1) then S
else let
   a = S[rand(#S)];
   S1 = {e in S | e < a};
   S2 = {e in S | e = a};
   S3 = {e in S | e > a};
   R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];
```

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

Quicksort Example

```
function quicksort(S) =
if (#S <= 1) then S
    else let a = S[rand(#S)];
    lesser = {e in S | e < a};
    equal = {e in S | e = a};
    greater = {e in S | e > a};
    R = {quicksort(v) : v in [lesser, greater]};
in R[0] ++ equal ++ R[1];
```

Example : Representing Graphs



Edge List Representation:

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

Adjacency List Representation:

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

Slide: Blelloch “NESL Revisited”, Intel Workshop 2006

Example : Graph Connectivity

L = Vertex Labels, E = Edge List

Use hashing to avoid non-determinism

```
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)

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

Lesson 1: Sequential Semantics

- Debugging is much easier without non-determinism
- Analyzing correctness is much easier without non-determinism
- If it works on one implementation, it works on all implementations
- Some problems are inherently concurrent—these aspects should be separated

Slide: Blelloch “NESL Revisited”, Intel Workshop 2006

Lesson 2: Cost Semantics

- Need a way to analyze cost, at least approximately, without knowing details of the implementation
- Any cost model based on processors is not going to be portable – too many different kinds of parallelism

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

Lesson 3: Too Much Parallelism

Needed ways to back out of parallelism

- Memory problem
- The "flattening" compiler technique was too aggressive on its own
- Need for Depth First Schedules or other scheduling techiques
- Various bounds shown on memory usage

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

Limitations

Communication was a bottleneck on machines available in the mid 1990s and required "micromanaging" data layout for peak performace.

Language would needs to be extended

- PSCICO Project (Parallel Scientific Computing) was looking into this

Hard to get users for a new language

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

Relevance to Multicore Architecture

- Communication is hopefully better than across chips
- Can make use of multiple forms of parallelism (multiple threads, multiple processors, multiple function units)
- Schedulers can take advantage of shared caching [SPAA04]
- Aggregate operations can possibly make use of on-chip hardware support

Slide: Blelloch "NESL Revisited", Intel Workshop 2006

NESL Overview

Syntax	Example
FUNCTION name(args) = exp ;	FUNCTION double(a) = 2*a;
IF e1 THEN e2 ELSE e3	IF (a > 22) THEN a ELSE 5*a
LET binding* IN exp	LET a = b*6; IN a + 3
{e1 : pattern IN e2}	{a + 22 : a IN [2, 1, 9]}
{pattern IN e1 | e2}	{a IN [2, 1, 9] | a < 8}
{e1 : p1 IN e2 ; p2 in e3}	{a + b : a IN [2,1]; b IN [7,11]}

Scalar Functions	
logical	not or and xor nor nand
comparison	== /= < > <= >=
predicates	plusp minusp zerop oddp evenp
arithmetic	+ - * / rem abs max min lshift rshift sqrt isqrt ln log exp expt sin cos tan asin acos atan sinh cosh tanh
conversion	btoi code_char char_code float ceil floor trunc round
random numbers	rand rand_seed
constants	pi max_int min_int