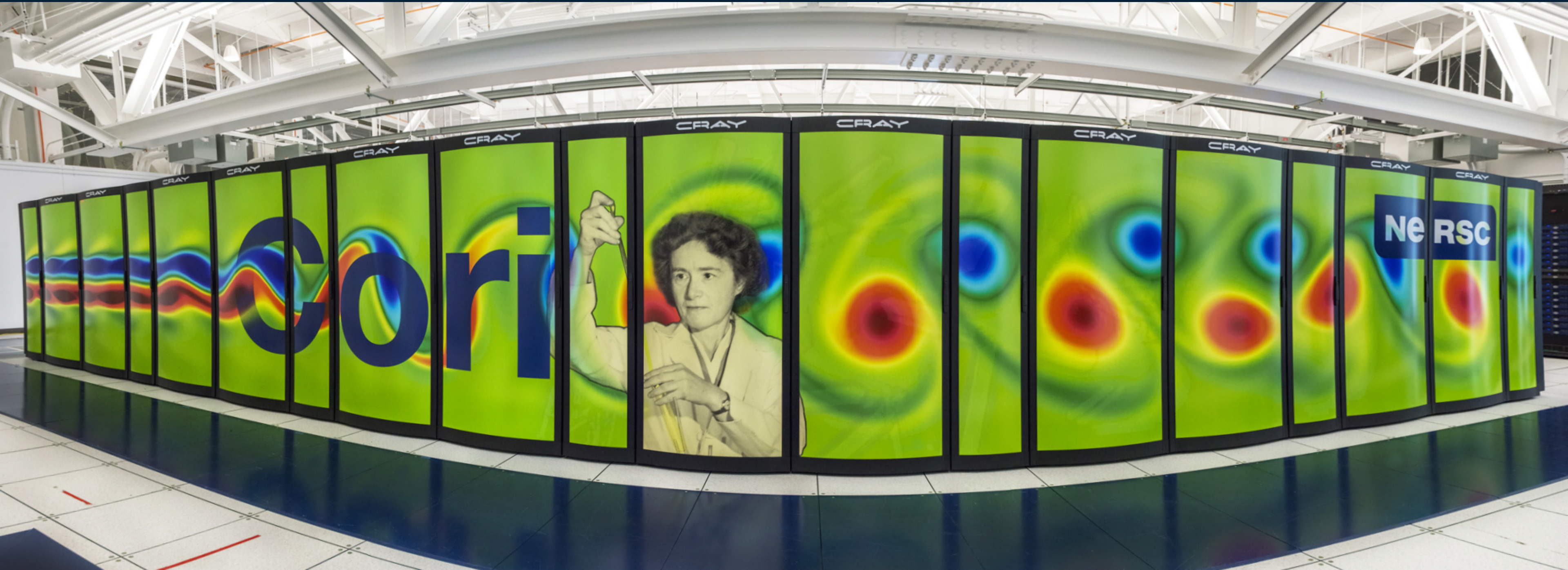
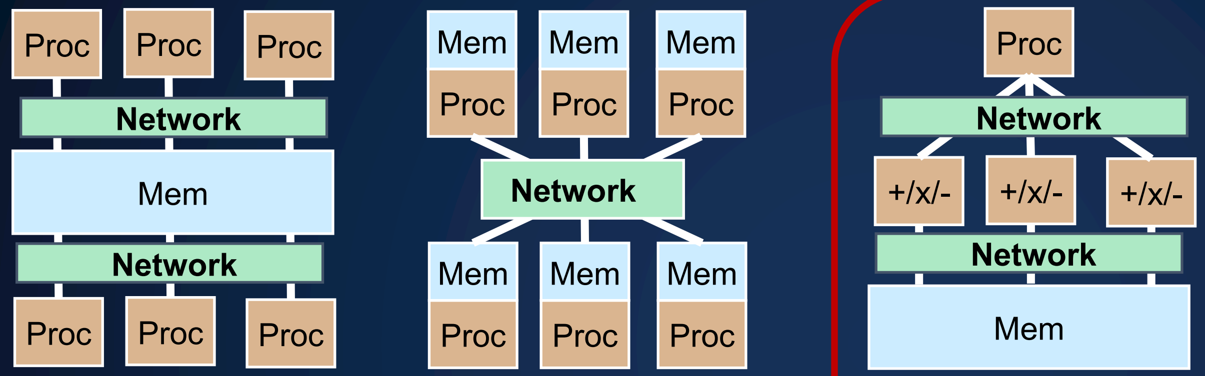
Applications of Parallel Computers

Data Parallel Algorithms

https://sites.google.com/lbl.gov/cs267-spr2021



Parallel Machines and Programming



Abstract Machine Models

Shared Memory	**Distributed Memory**	**Single Instruction Multiple Data (SIMD)**
Processors execute own instruction stream	Processors execute own instruction stream	One instruction stream (all run same instruction)
Communicate by reading/writing memory	Communicate by sending messages	Communicate through memory

The Power of Data Parallelism

- Data parallelism: perform the same operation on multiple values (often array elements)
  - Also includes reductions, broadcast, scan..
- Many parallel programming models use some data parallelism
  - SIMD units (and previously SIMD supercomputers)
  - CUDA / GPUs
  - MapReduce
  - MPI collectives

Data Parallel Programming: Unary Operators

- Unary operations applied to all elements of an array

A = array
B = array
f = square (any unary function, i.e., 1 argument)
B = f(A)

A:

3	1	1	2	3	3	4	2	2	2	1	3	1	1	1	3	3	2	1

f applied to each element

B:

9	1	1	4	9	9	16	4	4	4	1	9	1	1	1	9	9	4	1

Data Parallel Programming: Binary Operators

- Binary operations applied to all pairs of elements

A = array
B = array
C = array
- or any other binary operator
C = A - B

A:	3	1	0	2	3	0	4	2	0	2	1	3	0	1	1	0	3	2	1
B:	0	1	1	4	1	0	2	1	4	3	1	0	1	1	2	3	5	3	2
C:	3	0	-1	-2	2	0	2	2	-4	-2	0	3	-1	0	-1	-3	-2	-1	-1

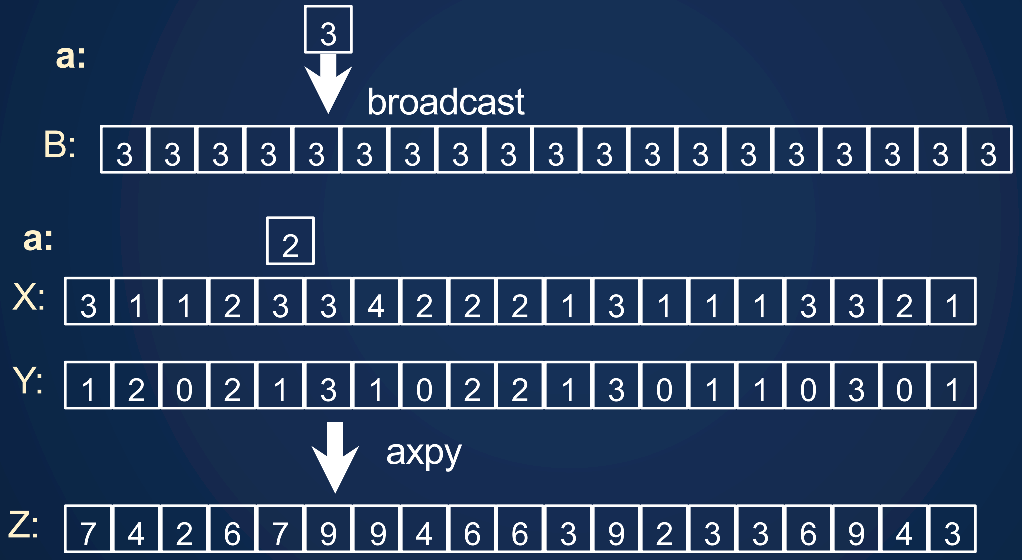
\- applied to each pair

Data Parallel Programming: Broadcast

- Broadcast fill a value into all elements of an array

a = scalar
B = a

a*X + Y



- Useful for a*X+Y called axpy, saxpy, daxpy
  - For single, double precision, or in general

Memory Operations: Strided and Scatter / Gather

- Array assignment works if the arrays are the same shape

```
A: double [0:4]
B: double [0:4] = [0.0, 1.0, 2.2, 3.3, 4.4]

A = B
```

Memory Operations: Strided and Scatter / Gather

- Array assignment works if the arrays are the same shape

```
A: double [0:4]
B: double [0:4] = [0.0, 1.0, 2.2, 3.3, 4.4]

A = B
```

- May have a stride, i.e., not be contiguous in memory

```
A = B [0:4:2]    // copy with stride 2 (every other element)
C: double [0:4, 0:4]
A = C [*,3]      // copy column of C
```

Memory Operations: Strided and Scatter / Gather

- Array assignment works if the arrays are the same shape

```
A: double [0:4]
B: double [0:4] = [0.0, 1.0, 2.2, 3.3, 4.4]

A = B
```

- May have a stride, i.e., not be contiguous in memory

```
A = B [0:4:2]    // copy with stride 2 (every other element)
C: double [0:4, 0:4]
A = C [*,3]      // copy column of C
```

- Gather (indexed) values from one array

```
X: int [0:4] = [3, 0, 4, 2, 1] // a permutation of indices 0 to 4
A = B[X]      // A now is [3.3, 0.0, 4.4, 2.2, 1.1]
```

Memory Operations: Strided and Scatter / Gather

- Array assignment works if the arrays are the same shape

```
A: double [0:4]
B: double [0:4] = [0.0, 1.0, 2.2, 3.3, 4.4]

A = B
```

- May have a stride, i.e., not be contiguous in memory

```
A = B [0:4:2]    // copy with stride 2 (every other element)
C: double [0:4, 0:4]
A = C [*,3]      // copy column of C
```

- Gather (indexed) values from one array

```
X: int [0:4] = [3, 0, 4, 2, 1] // a permutation of indices 1- 4
A = B[X]      // A now is [3.3, 0.0, 4.4, 2.2, 1.1]
```

- Scatter (indexed) values from one array

```
A[X] = B      // A now is [1.1, 4.4, 3.3, 0.0, 2.2]
```

Memory Operations: Strided and Scatter / Gather

- Array assignment works if the arrays are the same shape

```
A: double [0:4]
B: double [0:4] = [0.0, 1.0, 2.2, 3.3, 4.4]

A = B
```

- May have a stride, i.e., not be contiguous in memory

```
A = B [0:4:2]    // copy with stride 2 (every other element)
C: double [0:4, 0:4]
A = C [*,3]      // copy column of C
```

- Gather (indexed) values from one array

```
X: int [0:4] = [3, 0, 4, 2, 1] // a permutation of indices 1- 4
A = B[X]      // A now is [3.3, 0.0, 4.4, 2.2, 1.1]
```

- Scatter (indexed) values from one array

```
A[X] = B      // A now is [1.1, 4.4, 3.3, 0.0, 2.2]
```

What if X = [0,0,0,0,0]?

Data Parallel Programming: Masks

- Can apply operations under a “mask”

M = array of 0/1 (True/False)
A = array
B = array

A = A + B under M

A:	3	1	1	2	3	3	4	2	2	2	1	3	1	1	1	3	3	2	1
B:	0	1	1	4	1	0	2	1	4	3	1	0	1	1	2	3	5	3	2
M:	1	0	0	1	1	0	0	0	1	1	1	1	0	0	1	0	0	1	0

+ under mask

	0			4	1				4	3	1	0			2			3	
A:	3	1	1	6	4	3	4	2	6	5	2	3	1	1	3	3	3	5	1

Data Parallel Programming: Reduce

- Reduce an array to a value with + or any associative op

A = array
b = scalar
b = sum(A)

A:

3	1	1	2	3	3	4	2	2	2	1	3	1	1	1	3	3	2	1

sum reduction

b: 39

- Associative so we can perform op in different order
- Useful for dot products (ddot, sdot, etc.)

$b = X^T Y = \Sigma_j X[j] * Y[j]$

$b = dot(X, Y) = sum(X .* Y)$

X:

1	1	1	3	3	2	1

Y:

1	2	0	2	1	3	1

intermediate products

1	2	0	6	3	6	1

dot product

b: 19

Data Parallel Programming: Scans

- Fill array with partial reductions any associative op
- Sum scan:

A = array
B = array
B = scan(A,+)

A:

3	1	1	2	3	3	4	2	2	2	1	3	1	1	1	3	3	2	1

↓ sum scan

B:

3	4	5	7	10	13	17	19	21	23	24	27	28	29	30	33	36	38	39

- Max scan:

B = scan(A,max)

A:

3	1	1	2	3	3	4	2	2	2	1	3	1	1	1	3	3	2	1

↓ max scan

B:

3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	4	4

Inclusive and Exclusive Scans

Two variations of a scan, given an input vector $[x_0, x_1, \ldots, x_{n-1}]$:

- *inclusive* scan includes input x_i when computing output y_i

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

  e.g., add_scan_inclusive([1, 0, 3, 0, 2]) → [1, 1, 4, 4, 6]

- *exclusive* scan does *not* x_i when computing output y_i

  $[I, a_0, (a_0 ⊚ a_1), \ldots, (a_0 ⊚ a_1 \ldots ⊚ a_{n-2})]$ where I is the identity for ⊚

  e.g., add_scan_exclusive([1, 0, 3, 0, 2]) → [0, 1, 1, 4, 4]

Inclusive and Exclusive Scans

Two variations of a scan, given an input vector $[x_0, x_1, \ldots, x_{n-1}]$:

- *inclusive* scan includes input x_i when computing output y_i

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

  e.g., add_scan_inclusive([1, 0, 3, 0, 2]) → [1, 1, 4, 4, 6]

- *exclusive* scan does *not* x_i when computing output y_i

  $[I, a_0, (a_0 \circledcirc a_1), \ldots, (a_0 \circledcirc a_1 \ldots \circledcirc a_{n-2})]$ where I is the identity for $\circledcirc$

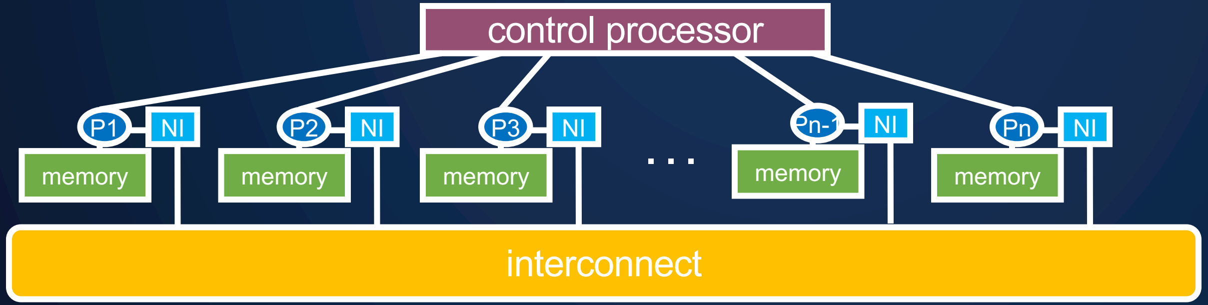
  e.g., add_scan_exclusive([1, 0, 3, 0, 2]) → [0, 1, 1, 4, 4]

Can easily get the inclusive version from the exclusive:
scan_inclusive(X) = X $\circledcirc$ scan_exclusive(X).
For the other way you need an inverse for $\circledcirc$ (or shift)

Idealized Hardware and Performance Model

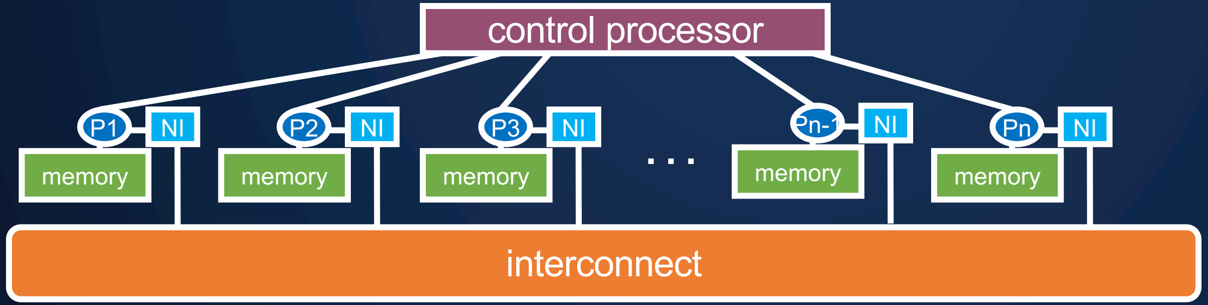
SIMD Systems Implemented Data Parallelism

- SIMD Machine: A large number of (usually) tiny processors.
  - A single "control processor" issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.



Ideal Cost Model for Data Parallelism

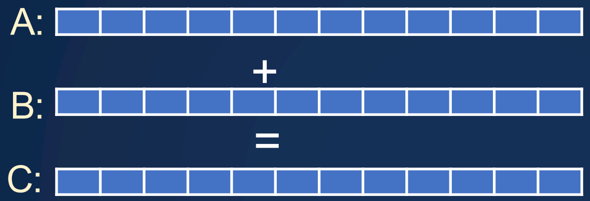
- Machine
  - An unbounded number of processors (p)
  - Control overhead is free
  - Communication is free
- Cost (complexity) on this abstract machine is the algorithm's *span or depth,* T_∞
  - Defines a lower bound on time on real machines



Cost on Ideal Machine (Span)

- Span for unary or binary operations (pleasingly parallel)

C = A+B

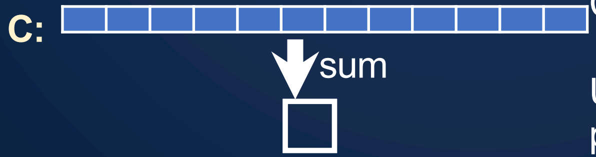


Cost O(1)
since p is unbounded

- Even if arrays are not aligned, communication is “free” here

- Reductions and broadcasts

s = sum(C)

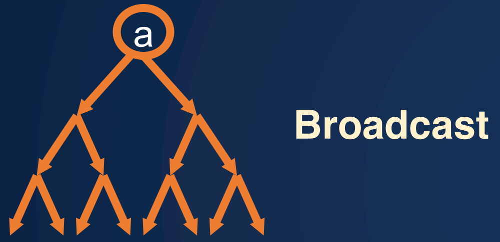


Cost O(log(n))

Using a tree of processors

Broadcast and reduction on processor tree

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



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

Can reductions go faster? No, log n lower bound

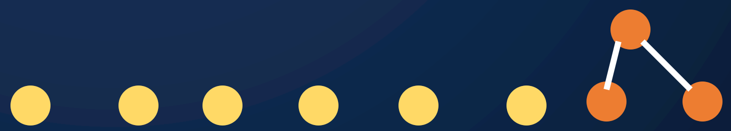
on any function of n variables!

n “useful” inputs 

Can reductions go faster? No, log n lower bound

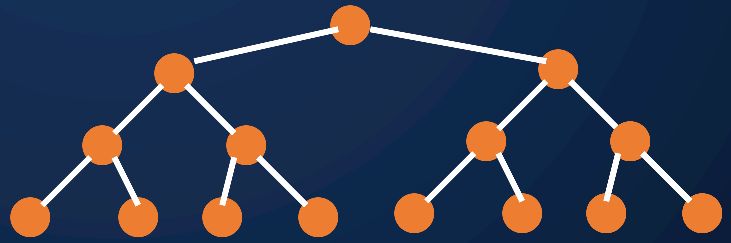
on any function of n variables!

- Given a function f (x1,…xn) of n input variables and 1 output variable, how fast can we evaluate it in parallel?
- Assume we only have binary operations, one per time step
- After 1 time step, an output can only depend on two inputs



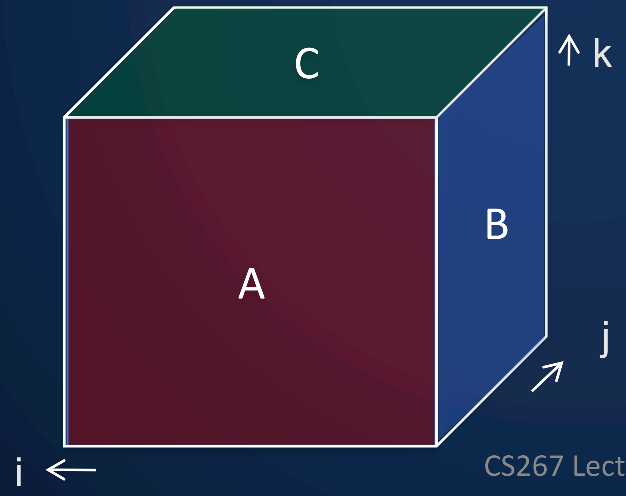
Can reductions go faster? No, log n lower bound on any function of n variables!

- Given a function f (x1,…xn) of n input variables and 1 output variable, how fast can we evaluate it in parallel?
- Assume we only have binary operations, one per time step
- After 1 time step, an output can only depend on two inputs
- By induction: after k time units, an output can only depend on 2^k inputs
  - After $\log_2$ n time units, output depends on at most n inputs
- A binary tree performs such a computation



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

- Use n^3 processors
- Step 1: For all ($1 <= i,j,k <= n$) $P(i,j,k) = A(i,k) * B(k,j)$
  - cost = 1 time unit, using n^3 processors
- Step 2: For all ($1 <= i,j <= n$) $C(i,j) = \sum_{k=1}^{n} P(i,j,k)$
  - cost = O(log n) time, using n^2 trees, $n^3 / 2$ processors each



Put a processor at every point in this cube

What about Scan (aka Parallel Prefix)?

- Recall: the scan 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

  $[a_0, (a_0 ◎ a_1), \ldots (a_0 ◎ a_1 \quad \ldots ◎ a_{n-1})]$
- Example: add scan of

  [1, 2, 0, 4, 2, 1, 1, 3] is [1, 3, 3, 7, 9, 10, 11, 14]
- Other operators
  - Reals: +, *, min, max (in floating point will assume associative)
  - Booleans: and, or
  - Matrices: mat mul

Can we parallelize a scan?

- It looks like this:

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

- Takes n-1 operations (adds) to do in serial

Can we parallelize a scan?

- It looks like this:

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

- Takes n-1 operations (adds) to do in serial
- The i^{th} iteration of the loop depends completely on the $(i-1)^{st}$ iteration.

- Impossible to parallelize, right?

A clue

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

What if we add, say, 5+6+7+8?

Parallel But Terribly Inefficient

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

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

Put 1 processor at element 1, 2 at element 2, 3 at position 3 …

- O(log n) span ☺
- O(n^2) work ☹

A clue

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

Is there any value in adding, say, 5+6+7+8?

If we separately have 1+2+3+4, what can we do?

A clue

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

Is there any value in adding, say, 5+6+7+8?

If we separately have 1+2+3+4, what can we do?

Suppose we added 1+2, 3+4, etc. pairwise, is this useful?

Sum Scan (aka prefix sum) in parallel

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

Sum Scan (aka prefix sum) in parallel

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

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

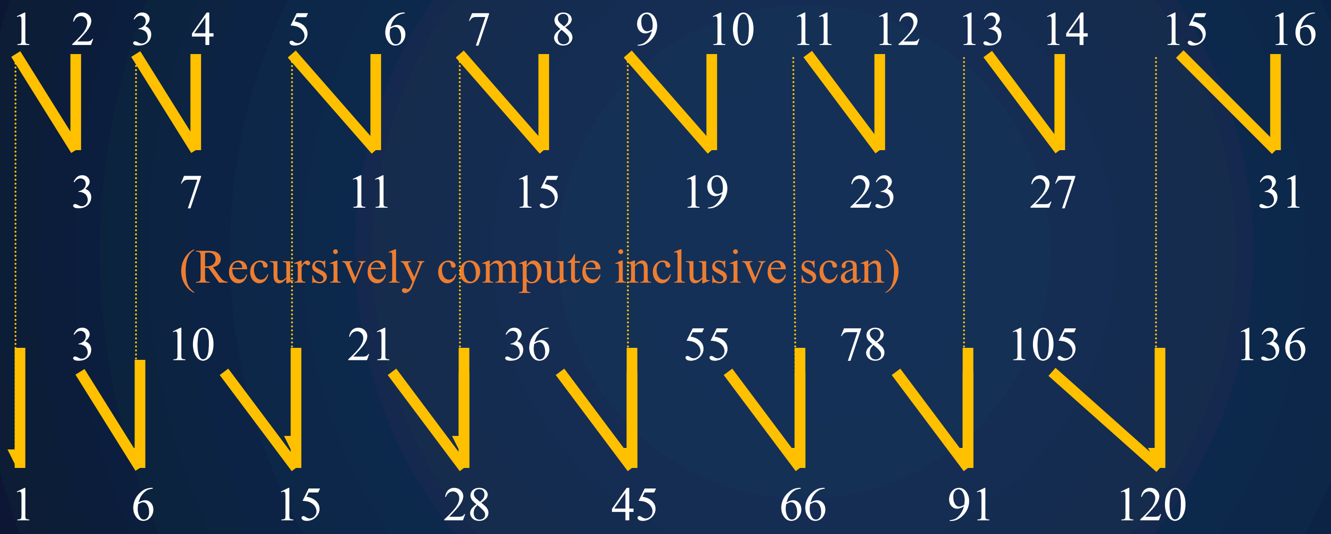
3 7 11 15 19 23 27 31

(Recursively compute inclusive scan)

3 10 21 36 55 78 105 136

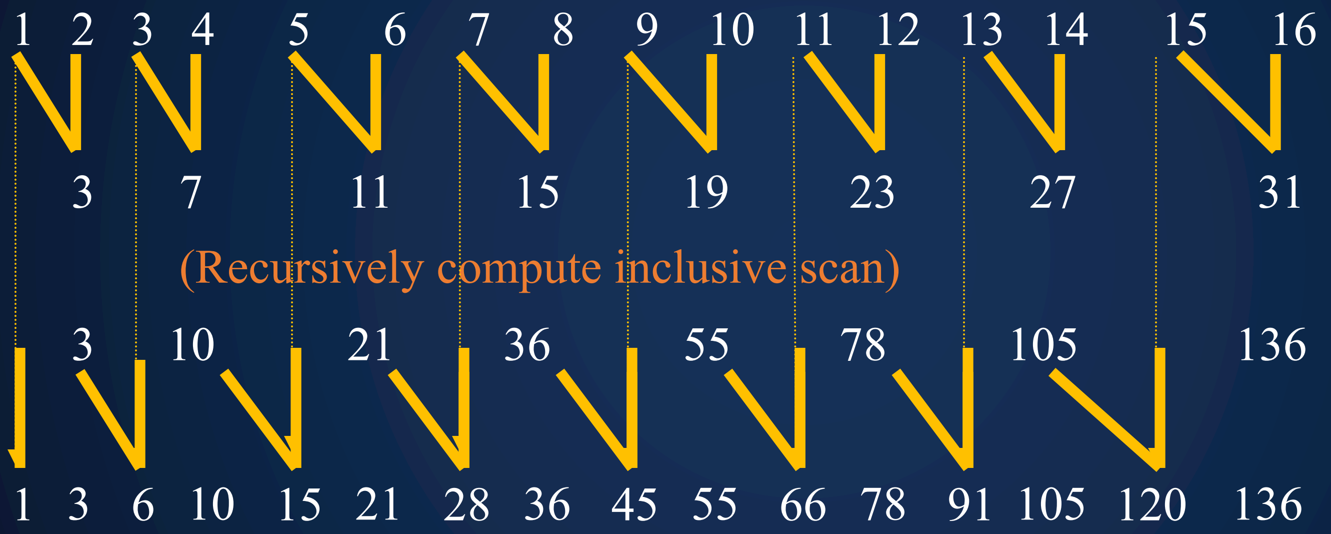
Sum Scan (aka prefix sum) in parallel

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



Sum Scan (aka prefix sum) in parallel

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



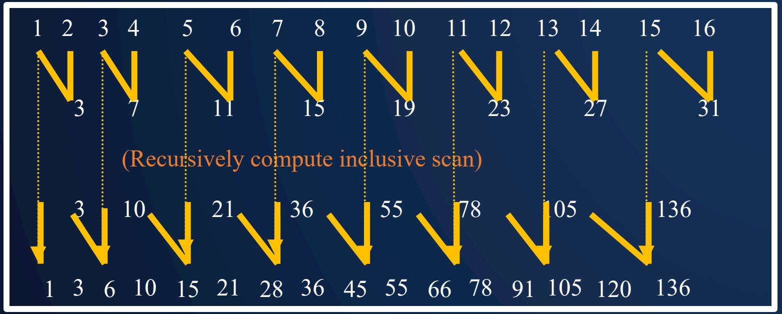
Parallel prefix cost

Time for this algorithm on one processor (work)

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

Time on unbounded number of processors (span)

- $T_\infty(n) = 2 \log n$



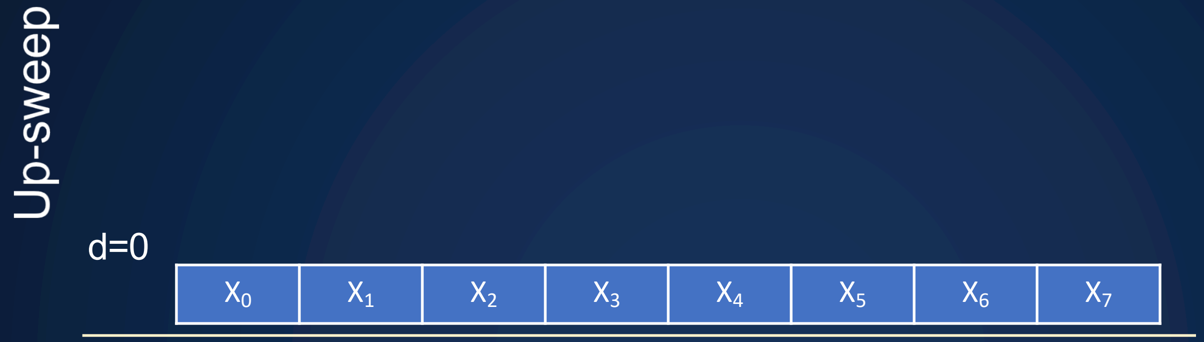
Pairwise sum

Recursive prefix

Pairwise sum (update odds)

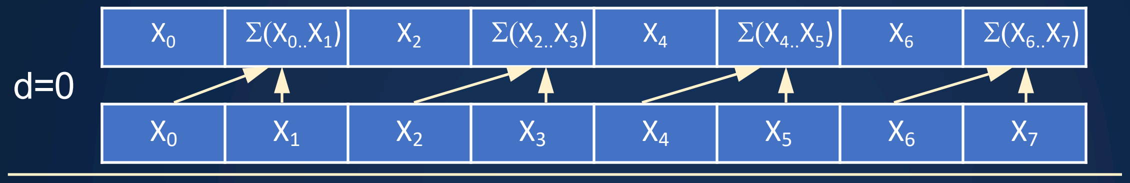
Parallelism at the cost of more work (2x)!

Non-recursive exclusive scan



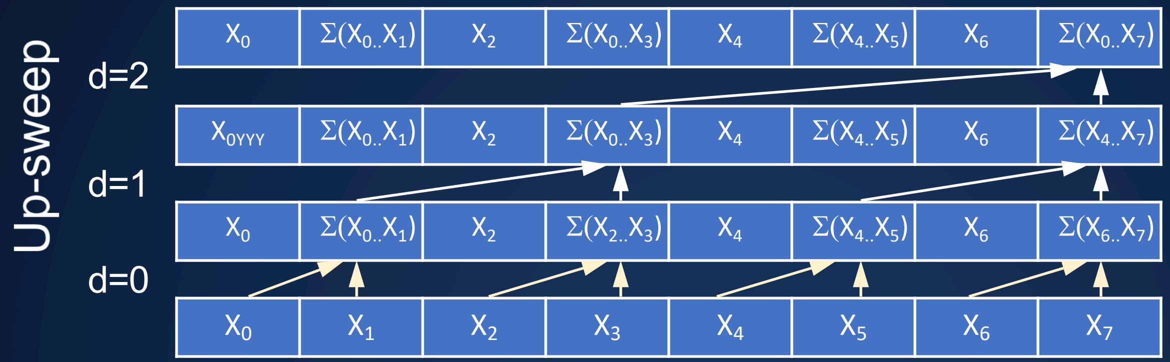
Algorithm due to Blelloch

Non-recursive exclusive scan



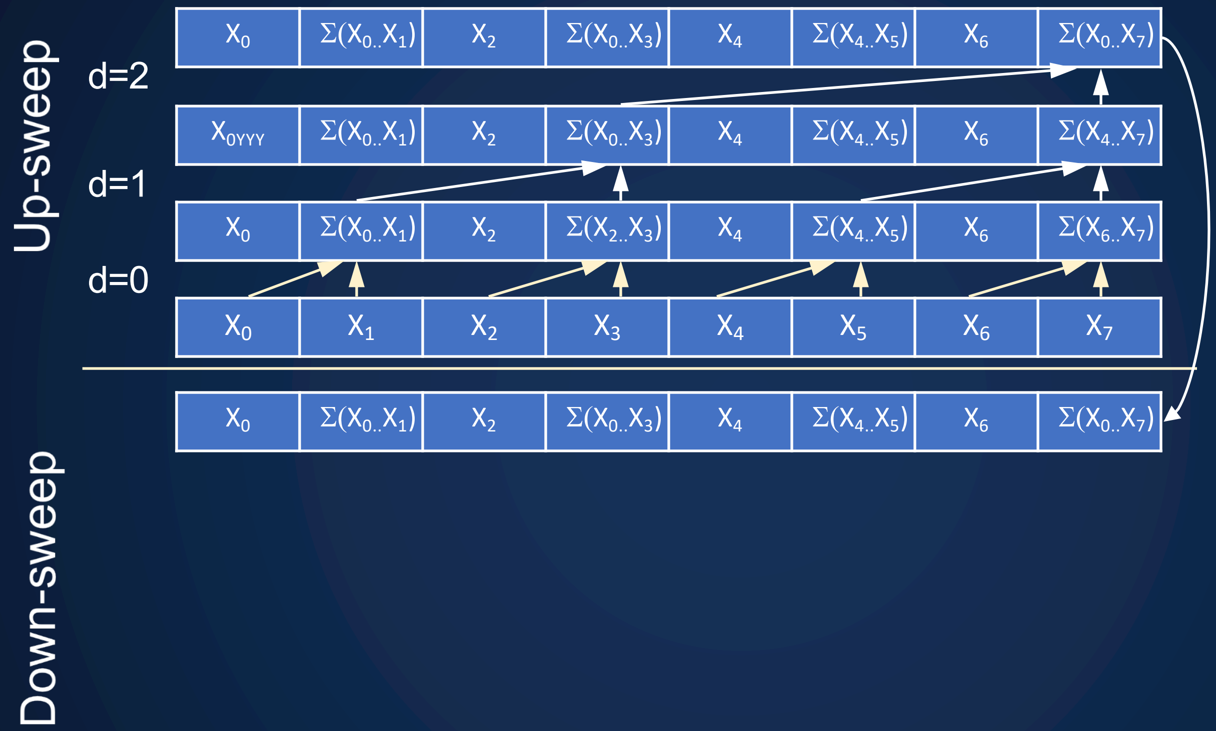
Algorithm due to Blelloch

Non-recursive exclusive scan



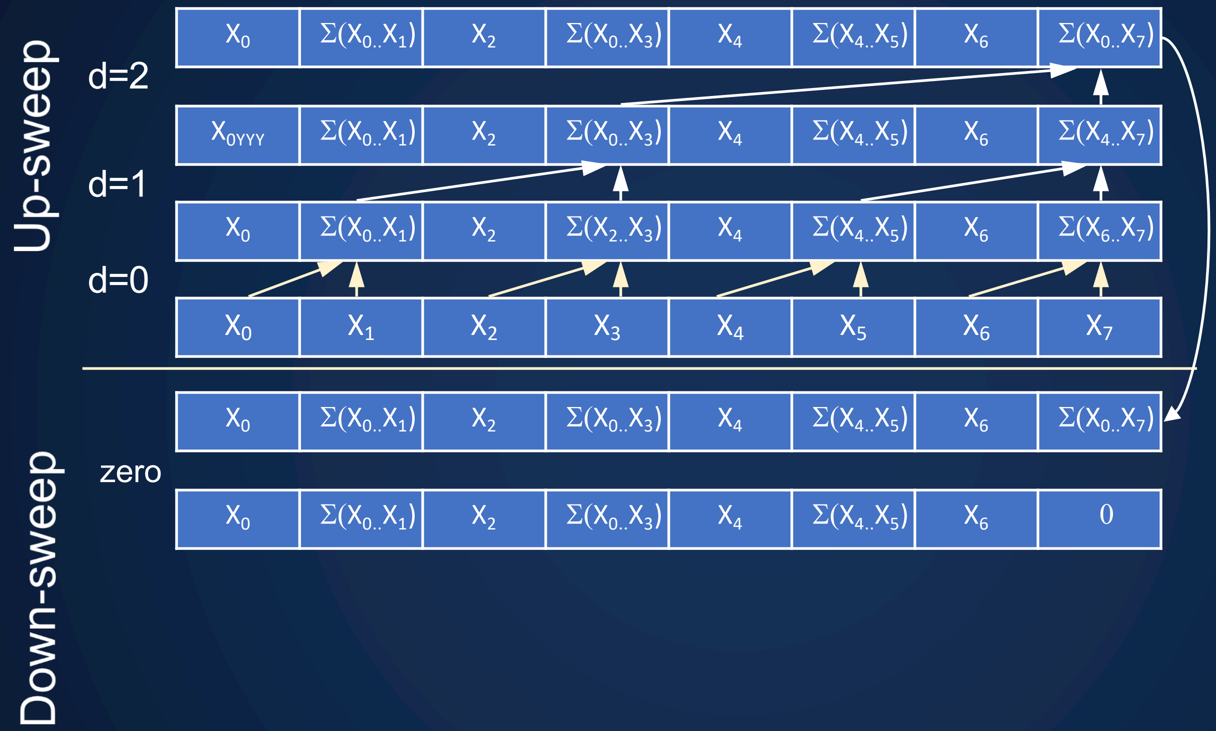
Algorithm due to Blelloch

Non-recursive exclusive scan



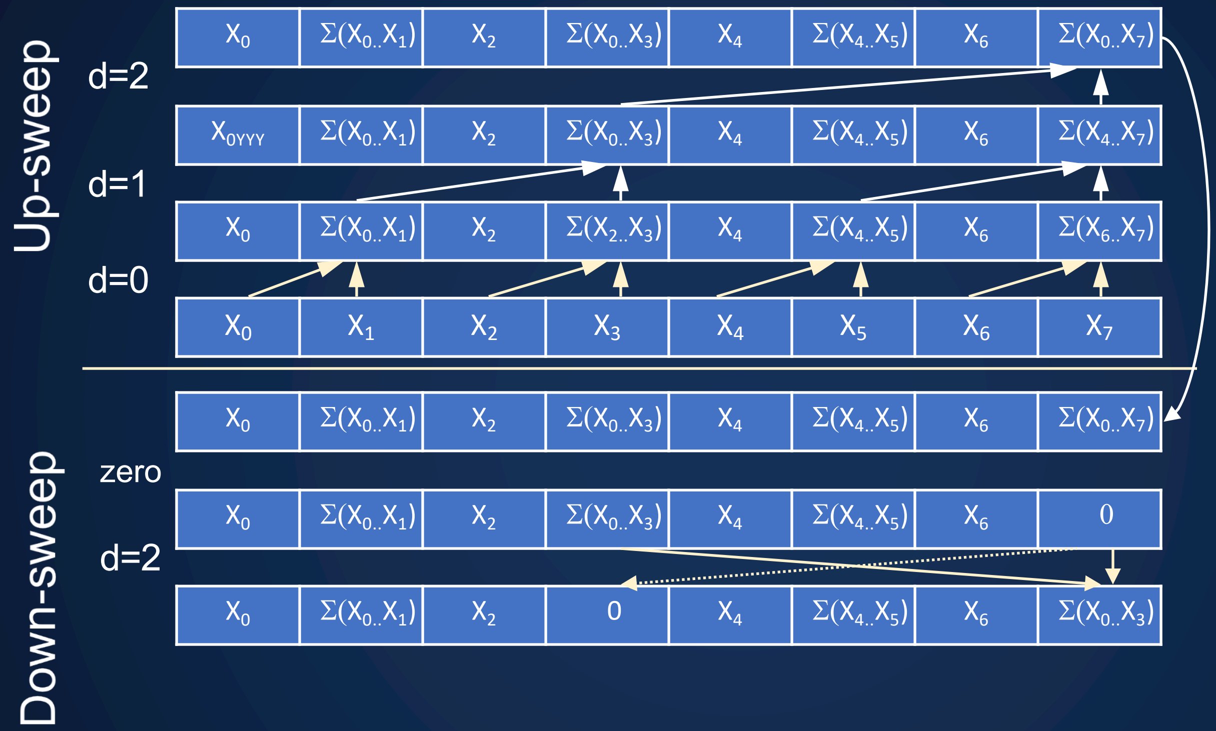
Algorithm due to Blelloch

Non-recursive exclusive scan



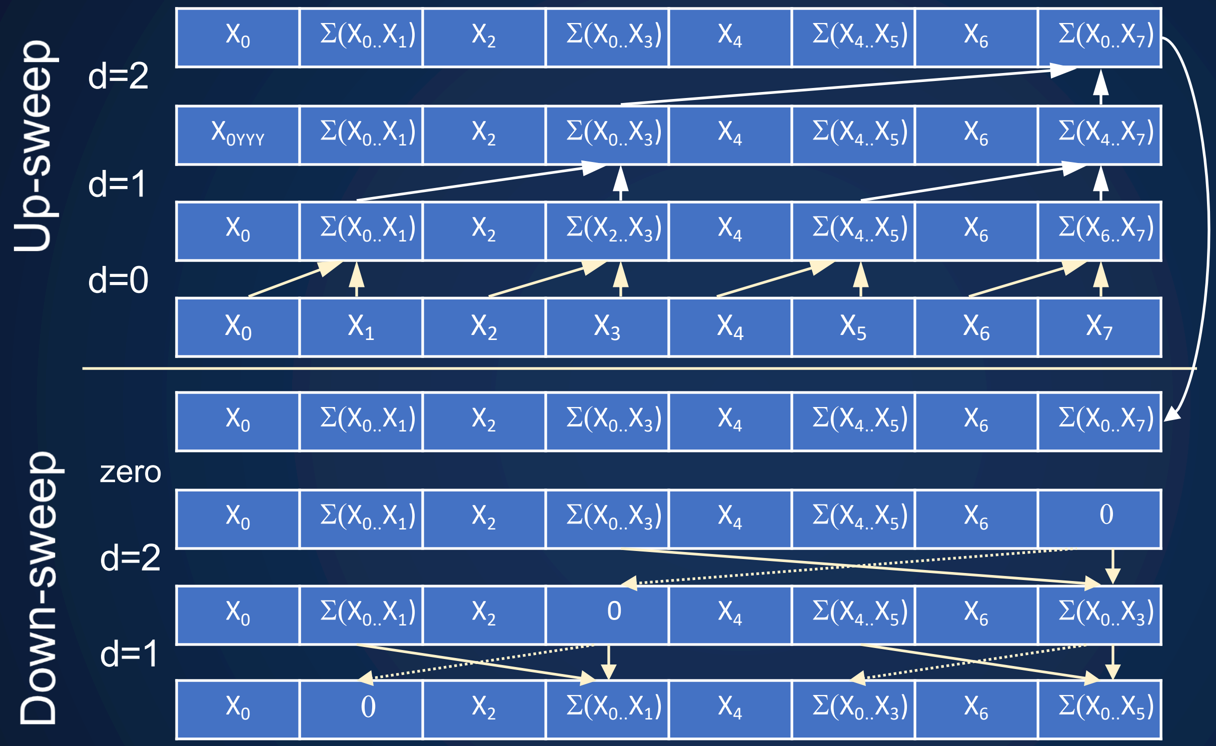
Algorithm due to Blelloch

Non-recursive exclusive scan



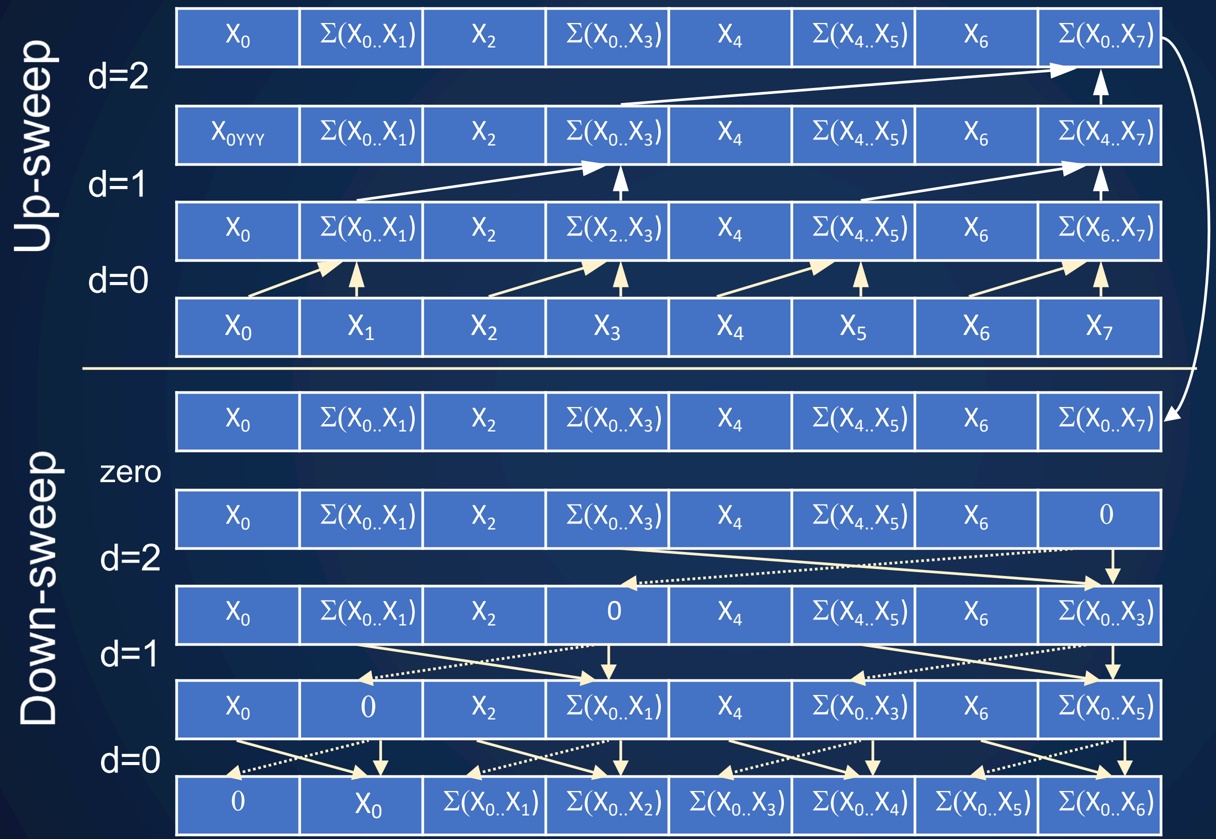
Algorithm due to Blelloch

Non-recursive exclusive scan



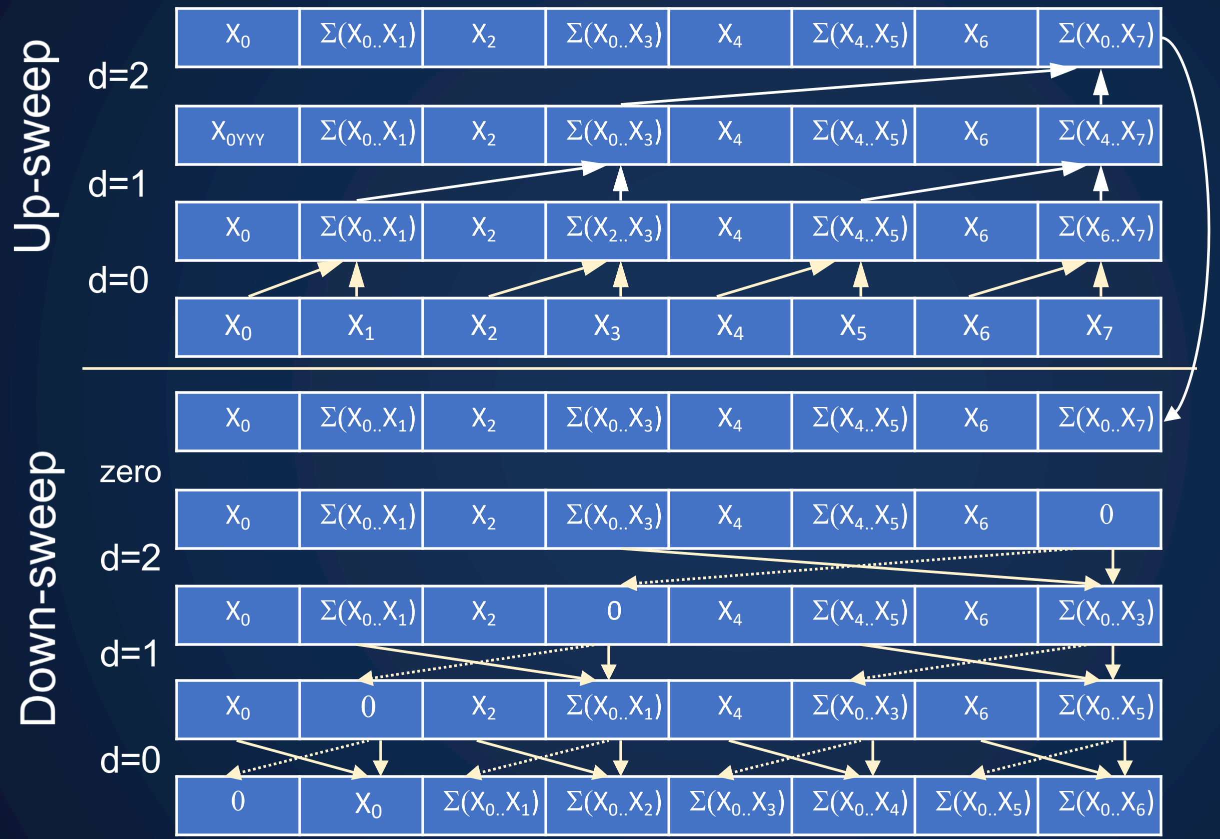
Algorithm due to Blelloch

Non-recursive exclusive scan



Algorithm due to Blelloch

Non-recursive exclusive scan



This is both work-efficient (n adds) and space-efficient (update in place)

Algorithm due to Blelloch

(Non-trivial) Applications of Data Parallelism

..using scans

Scans are useful for many things (partial list here)

- 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…

Scans are useful for many things (partial list here)

- 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…

Application: Stream Compression

- Given an array of 0/1 flags

flags =

1	0	1	1	0	0	1	1

and an array (stream) of values

values =

3	2	4	1	5	3	3	1

compress into

result =

3	4	1	3	1

Application: Stream Compression

- Given an array of 0/1 flags

flags =

1	0	1	1	0	0	1	1

and an array (stream) of values

values =

3	2	4	1	5	3	3	1

compress into

result =

3	4	1	3	1

- Step 1: Compute an exclusive add scan of flags:

index =

0	1	1	2	3	3	3	4

Application: Stream Compression

- Given an array of 0/1 flags

flags =

1	0	1	1	0	0	1	1

and an array (stream) of values

values =

3	2	4	1	5	3	3	1

compress into

result =

3	4	1	3	1

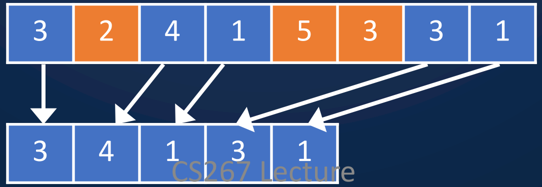
- Step 1: Compute an exclusive add scan of flags:

index =

0	1	1	2	3	3	3	4

- Step 2: “Scatter” values into result at index, masked by flags

result[index] = values at flags



Remove matching elements

- Given an array of values, and an int x, remove all elements that are not divisible by x:

  int find (int x, int y) (y % x == 0) ? 1 : 0;

values =

3	5	6	12	4	2	3	0

flags = apply(values, find)

1	0	1	1	0	0	1	1

Use previous solution to remove those not divisible

Application: Radix Sort (serial algorithm)

4	7	2	6	3	5	1	0

Idea: Sort 1 bit at a time:

- *0s on left, 1s on right*

Use a "stable" sort:

- *Keep order as it, unless things need to switch based on the current bit*

Start with least-significant bit

- *And move up*

Application: Radix Sort (serial algorithm)

4	7	2	6	3	5	1	0

Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Application: Radix Sort (serial algorithm)

4	7	2	6	3	5	1	0

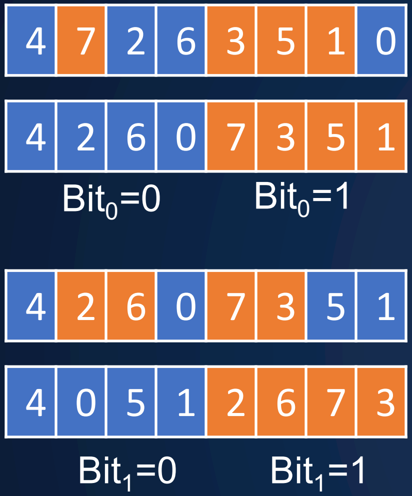
4	2	6	0	7	3	5	1

Bit_0=0 Bit_0=1

Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Application: Radix Sort (serial algorithm)



Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Stably sort entire array on next bit

X0X < X1X

Application: Radix Sort (serial algorithm)

4	7	2	6	3	5	1	0

Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

4	2	6	0	7	3	5	1

Bit_0=0 Bit_0=1

4	2	6	0	7	3	5	1

Stably sort entire array on next bit

X0X < X1X

Application: Radix Sort (serial algorithm)

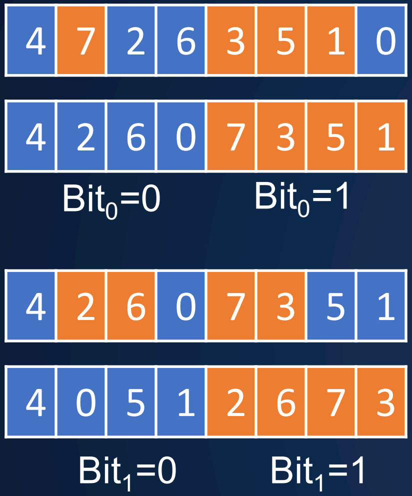
Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Stably sort entire array on next bit

X0X < X1X

Application: Radix Sort (serial algorithm)



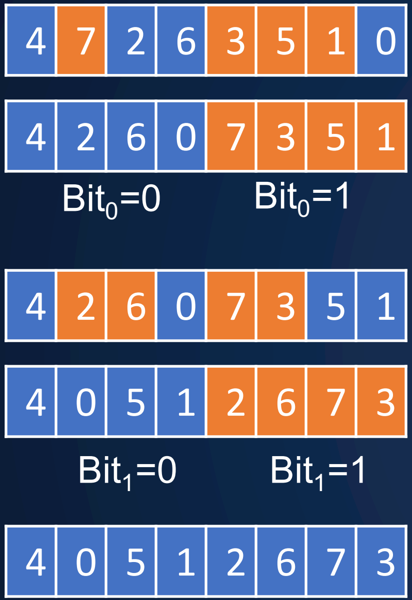
Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Stably sort entire array on next bit

X0X < X1X

Application: Radix Sort (serial algorithm)



Sort on least significant bit (Bit$_0$ in [Bit$_2$, Bit$_1$, Bit$_0$])

XX0 < XX1 (evens before odds)

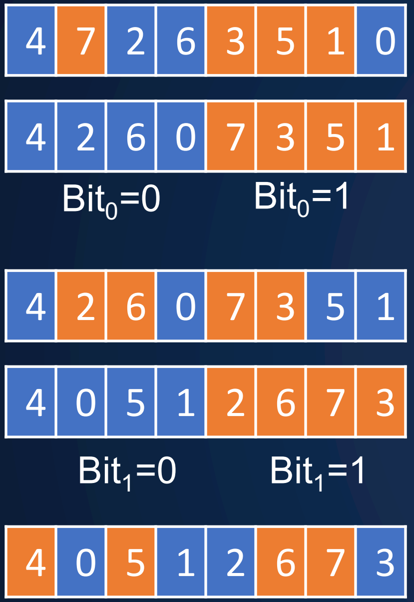
Stably sort entire array on next bit

X0X < X1X

Stably sort on next bit

0XX < 1XX (<4 before >=4 here)

Application: Radix Sort (serial algorithm)



Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Stably sort entire array on next bit

X0X < X1X

Stably sort on next bit

0XX < 1XX (<4 before >=4 here)

Application: Radix Sort (serial algorithm)

Sort on least significant bit (Bit_0 in [Bit_2, Bit_1, Bit_0])

XX0 < XX1 (evens before odds)

Stably sort entire array on next bit

X0X < X1X

Stably sort on next bit

0XX < 1XX (<4 before >=4 here)

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0

input

odds = last bit of each element

evens = complement of odds (last bit = 0)

evpos = exclusive sum scans of evens

totalEvens = broadcast last element

indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

This will just do one step of radix sort (a stable sort on 1 bit)

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0

input

0	1	0	0	1	1	1	0

odds = last bit of each element

evens = complement of odds (last bit = 0)

evpos = exclusive sum scans of evens

totalEvens = broadcast last element

indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0	input
0	1	0	0	1	1	1	0	odds = last bit of each element

evens = complement of odds (last bit = 0)

evpos = exclusive sum scans of evens

totalEvens = broadcast last element

indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0
0	1	0	0	1	1	1	0
1	0	1	1	0	0	0	1

input

odds = last bit of each element

evens = complement of odds (last bit = 0)

evpos = exclusive sum scans of evens

totalEvens = broadcast last element

indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens

totalEvens = broadcast last element

indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element

indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		indx = constant array of 0..n

oddpos = totalEvens + indx– epos

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		indx = constant array of 0..n
4	4	5	5	5	6	7	8		oddpos = totalEvens + indx– epos

4+0-0	4+1-1	4+2-1	4+3-2	4+4-3	4+5-3	4+6-3	4+7-3

pos = if evens then evpos else oddpos

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		indx = constant array of 0..n
4	4	5	5	5	6	7	8		oddpos = totalEvens + indx– epos
0	4	1	2	5	6	7	3		pos = if evens then evpos else oddpos

4+0-0	4+1-1	4+2-1	4+3-2	4+4-3	4+5-3	4+6-3	4+7-3

Scatter input using pos as index

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		indx = constant array of 0..n
4	4	5	5	5	6	7	8		oddpos = totalEvens + indx– epos
0	4	1	2	5	6	7	3		pos = if evens then evpos else oddpos
4	7	2	6	3	5	1	0		Scatter input using pos as index

4+0-0	4+1-1	4+2-1	4+3-2	4+4-3	4+5-3	4+6-3	4+7-3

Using two masked assignments

Repeat with next bit to left until done

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		indx = constant array of 0..n
4+0-0	4+1-1	4+2-1	4+3-2	4+4-3	4+5-3	4+6-3	4+7-3		
4	4	5	5	5	6	7	8		oddpos = totalEvens + indx– epos
0	4	1	2	5	6	7	3		pos = if evens then evpos else oddpos
4	7	2	6	3	5	1	0		Scatter input using pos as index
4	2	6	0	7	3	5	1		Repeat with next bit to left until done

Using two masked assignments

Application: Data Parallel Radix Sort

4	7	2	6	3	5	1	0		input
0	1	0	0	1	1	1	0		odds = last bit of each element
1	0	1	1	0	0	0	1		evens = complement of odds (last bit = 0)
0	1	1	2	3	3	3	3	4	evpos = exclusive sum scans of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		indx = constant array of 0..n
4+0-0	4+1-1	4+2-1	4+3-2	4+4-3	4+5-3	4+6-3	4+7-3		
4	4	5	5	5	6	7	8		oddpos = totalEvens + indx– epos
0	4	1	2	5	6	7	3		pos = if evens then evpos else oddpos
4	7	2	6	3	5	1	0		Scatter input using pos as index
4	2	6	0	7	3	5	1		Repeat with next bit to left until done

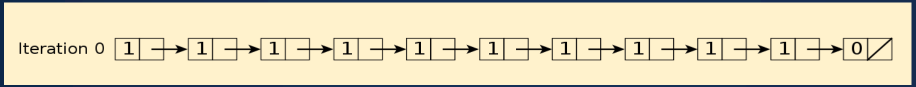
Using two masked assignments

List Ranking with Pointer Doubling

Given a linked list of *N* nodes, find the distance (#hops) from each node to the end of the list.

```
d(n)=
    0 if n.next is null
    1+d(n.next) otherwise
```

Approach: put a processor at every node



Steps:

```
val = 1
while next != null
  val += next.val
  next =
next.next
```

Works if nodes are on arbitrary processors

List Ranking with Pointer Doubling

Steps:

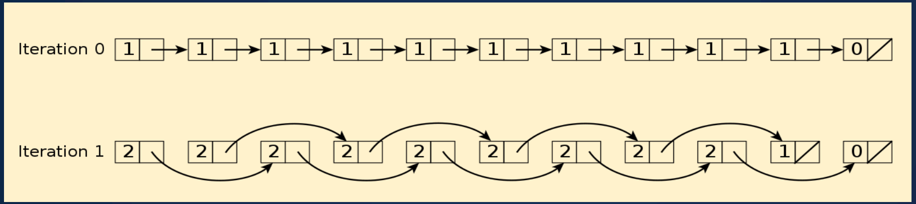
val = 1
while next != null
  val += next.val
  next = next.next

Works if nodes are on arbitrary processors

Given a linked list of *N* nodes, find the distance (#hops) from each node to the end of the list.

```
d(n)=
    0 if n.next is null
    1+d(n.next) otherwise
```

Approach: put a processor at every node



List Ranking with Pointer Doubling

Steps:

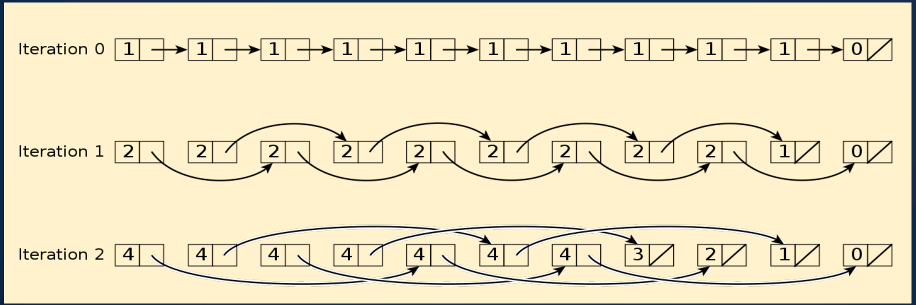
val = 1
while next != null
  val += next.val
  next = next.next

Works if nodes are on arbitrary processors

Given a linked list of *N* nodes, find the distance (#hops) from each node to the end of the list.

```
d(n)=
    0 if n.next is null
    1+d(n.next) otherwise
```

Approach: put a processor at every node



List Ranking with Pointer Doubling

Given a linked list of *N* nodes, find the distance (#hops) from each node to the end of the list.

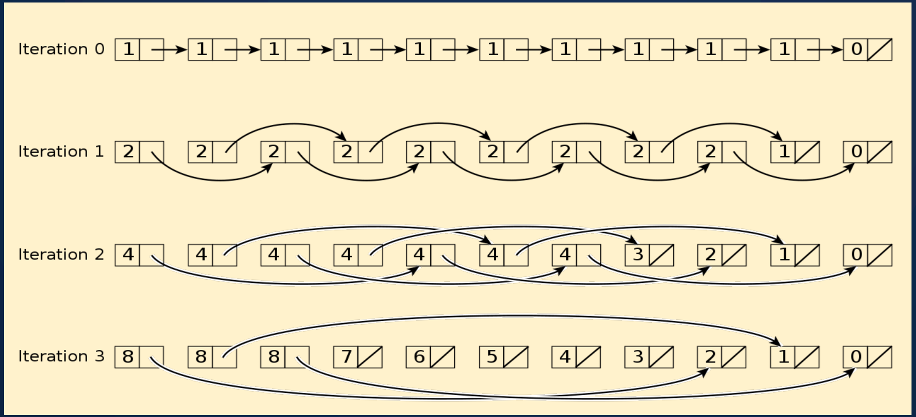
```
d(n)=
    0 if n.next is null
    1+d(n.next) otherwise
```

Approach: put a processor at every node

Steps:

val = 1
while next != null
  val += next.val
  next = next.next

Works if nodes are on arbitrary processors



List Ranking with Pointer Doubling

Given a linked list of *N* nodes, find the distance (#hops) from each node to the end of the list.

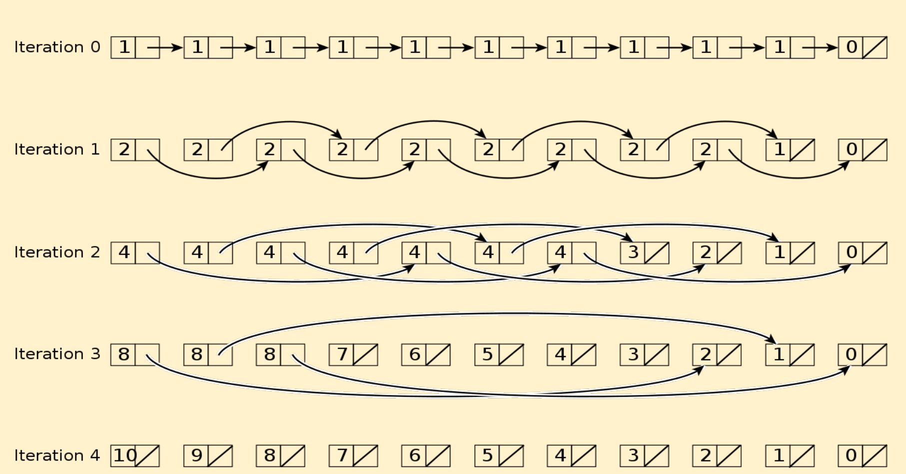
```
d(n)=
    0 if n.next is null
    1+d(n.next) otherwise
```

Approach: put a processor at every node

Steps:

val = 1
while next != null
  val += next.val
  next =
next.next

Works if nodes are on arbitrary processors



Application: Fibonacci via Matrix Multiply Prefix

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

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

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

Slide source: Alan Edelman

Application: Adding n-bit integers in O(log n) time

- Computing sum s of two n-bit binary numbers, think of a and b as array of bits
  - a = a[n-1] a[n-2]…a[0] and b = b[n-1] b[n-2]…b[0]
  - s = a+b = s[n] s[n-1]…s[0] (use carry-bit array c = c[n-1]…c[0] c[-1])

Application: Adding n-bit integers in O(log n) time

- Computing sum s of two n-bit binary numbers, a and b
  - a = a[n-1] a[n-2]…a[0] and b = b[n-1] b[n-2]…b[0]
  - s = a+b = s[n] s[n-1]…s[0] (use carry-bit array c = c[n-1]…c[0] c[-1])
- Formula c[-1] = 0 … rightmost carry bit
  for i = 0 to n-1 … compute right to left
  s[i] = (a[i] xor b[i]) xor c[i-1] … one or three 1s
  c[i] = ((a[i] xor b[i]) and c[i-1]) or (a[i] and b[i]) ... next carry bit

Application: Adding n-bit integers in O(log n) time

- Computing sum s of two n-bit binary numbers, a and b
  - a = a[n-1] a[n-2]…a[0] and b = b[n-1] b[n-2]…b[0]
  - s = a+b = s[n] s[n-1]…s[0] (use carry-bit array c = c[n-1]…c[0] c[-1])
- Formula c[-1] = 0 … rightmost carry bit
  for i = 0 to n-1 … compute right to left
  s[i] = (a[i] xor b[i]) xor c[i-1] … one or three 1s
  c[i] = ((a[i] xor b[i]) and c[i-1]) or (a[i] and b[i]) ... next carry bit
- Example
  - a = 22
  - b = 29

```
a =     1 0 1 1 0      (22)
b =     1 1 1 0 1      (29)
c =   1 1 1 0 0 0 0
s =   1 1 0 0 1 1      (51)
```

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

Application: Adding n-bit integers in O(log n) time

- Recall carry bit calculation

```
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
```

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

Application: Adding n-bit integers in O(log n) time

- Recall carry bit calculation

```
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
```

- 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
```

Both O(1) on n procs

Application: Adding n-bit integers in O(log n) time

- Recall carry bit calculation

      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

- 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

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

Application: Adding n-bit integers in O(log n) time

- Recall carry bit calculation

    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

- 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

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

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

Application: Adding n-bit integers in O(log n) time

- Recall carry bit calculation

  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

- 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

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

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

… evaluate M[i] * M[i-1] * … * M[0] by parallel prefix
… 2-by-2 Boolean matrix multiplication is associative

Application: Adding n-bit integers in O(log n) time

- Recall carry bit calculation

  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

- 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

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

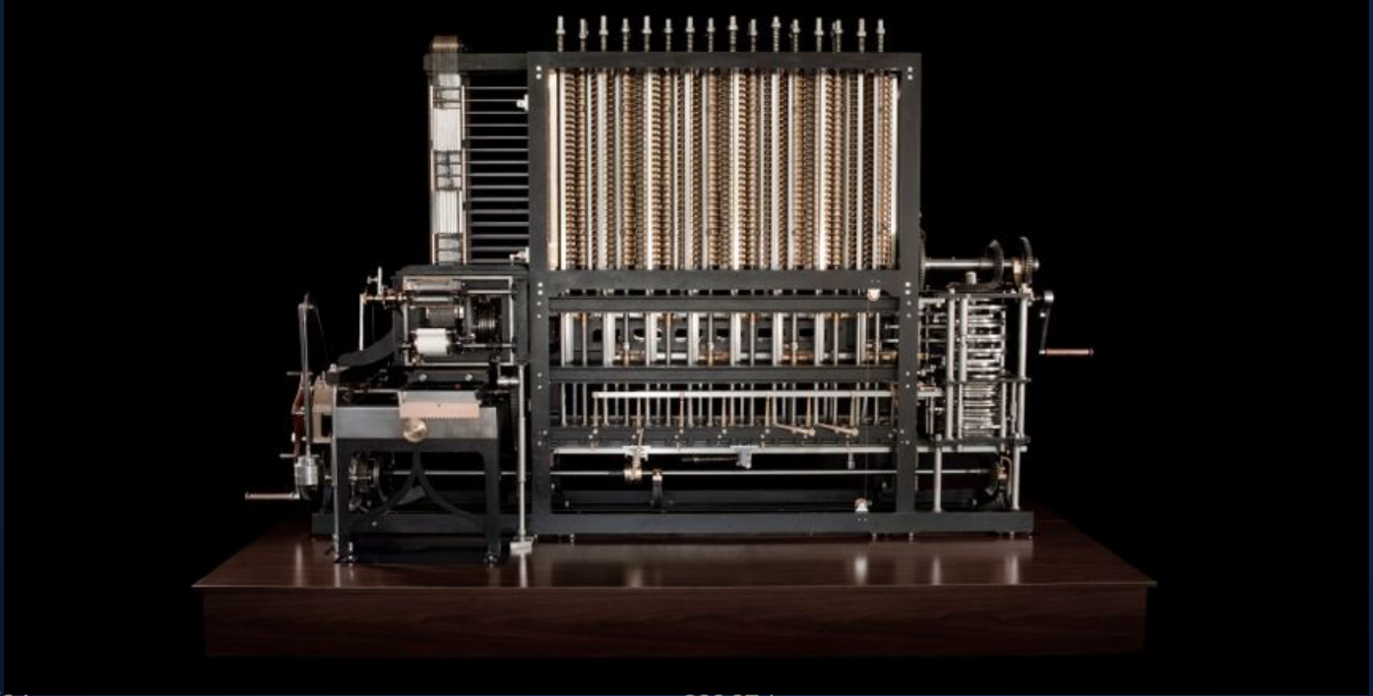
$$= M[i] * M[i-1] * \ldots M[0] * \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

… evaluate M[i] * M[i-1] * … * M[0] by parallel prefix
… 2-by-2 Boolean matrix multiplication is associative

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

This idea is used in all hardware

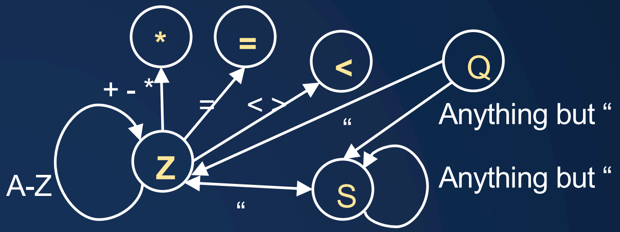
…Even going back to Babbage



Lexical analysis (tokenizing, scanning)

- Given a language of:
  - Identifiers (Z): string of chars
  - Strings (S): in double quotes
  - Ops (*): +,-,*,=,<,>,<=, >=
  - Expression (E), Quotes (Q),…
- Lexical analysis
  - Divide into tokens

Subset of Finite State Machine for Lexical Analysis



if	x	<=	n	then	print	”hello world”

Full finite state machine encoded in table

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

- Each state in first column; N initial state.
- Each row gives the next state based on the next character at the top.
- Apply string Y”+ to state Z written as ZY”+ = ((ZY)“)+ = (Z”)+ = Q+ = S
- Each column is a state transition for that character

Hillis and Steele, CACM 1986

Lexical analysis (tokenizing, scanning)

- 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 (N, row 1) to index into these arrays.

	i	f		x		<	=		n
N	A	A	N	A	N	<	*	N	A
A	Z	Z	N	Z	N	<	*	N	Z
Z	Z	Z	N	Z	N	<	*	N	Z
*****	A	A	N	A	N	<	*	N	A
<	A	A	N	A	N	<	=	N	A
=	A	A	N	A	N	<	*	N	A
Q	S	S	S	S	S	S	S	S	S
S	S	S	S	S	S	S	S	S	S
E	E	E	N	E	N	<	*	N	E

<= n
A
A
A
A
A
A
S
S
A

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

Hillis and Steele, CACM 1986

Inverting triangular n-by-n matrices

in $O(\log^2 n)$ time

- Fact: $$\begin{pmatrix} A & 0 \\ C & B \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ -B^{-1}CA^{-1} & B^{-1} \end{pmatrix}$$

Inverting triangular n-by-n matrices

in $O(\log^2 n)$ time

- Fact: $\begin{bmatrix} A & 0 \\ C & B \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -B^{-1}CA^{-1} & B^{-1} \end{bmatrix}$

- Function Tri_Inv(T) // assume n = dim(T) = 2^m for simplicity

```
if T is 1-by-1
    return 1/T
else                Write T = [A 0; C B]

in parallel do {
    invA = Tri_Inv(A)
    invB = Tri_Inv(B)       //  implicitly uses a tree
}
newC = -invB * C * invA  // log(n) for matmuls

return     [invA  0; newC  invB]
```

Write $T = \begin{bmatrix} A & 0 \\ C & B \end{bmatrix}$, return $\begin{bmatrix} invA & 0 \\ newC & invB \end{bmatrix}$

Inverting triangular n-by-n matrices

in $O(\log^2 n)$ time

- Fact: $$\begin{bmatrix} A & 0 \\ C & B \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -B^{-1}CA^{-1} & B^{-1} \end{bmatrix}$$

- Function Tri_Inv(T) // assume n = dim(T) = 2^m for simplicity

```
if T is 1-by-1
     return 1/T
else
```

Write $T = \begin{bmatrix} A & 0 \\ C & B \end{bmatrix}$

```
in parallel do {
     invA = Tri_Inv(A)
     invB = Tri_Inv(B)      //  implicitly uses a tree
}
newC = -invB * C * invA  // log(n) for matmuls
```

return $\begin{bmatrix} \text{invA} & 0 \\ \text{newC} & \text{invB} \end{bmatrix}$

time(Tri_Inv(n)) =
time(Tri_Inv(n/2)) + O(log(n))

Change variable to m = log n to get time(Tri_Inv(n)) = $O(\log^2 n)$

Segmented Scans

Inputs = value array, flag array,
associative operator ⊕

Inclusive segmented sum scan

1	2	3	4	5	6	7	8

0	0	1	0	0	1	0	1

Result

1	3	3	7	12	6	13	8

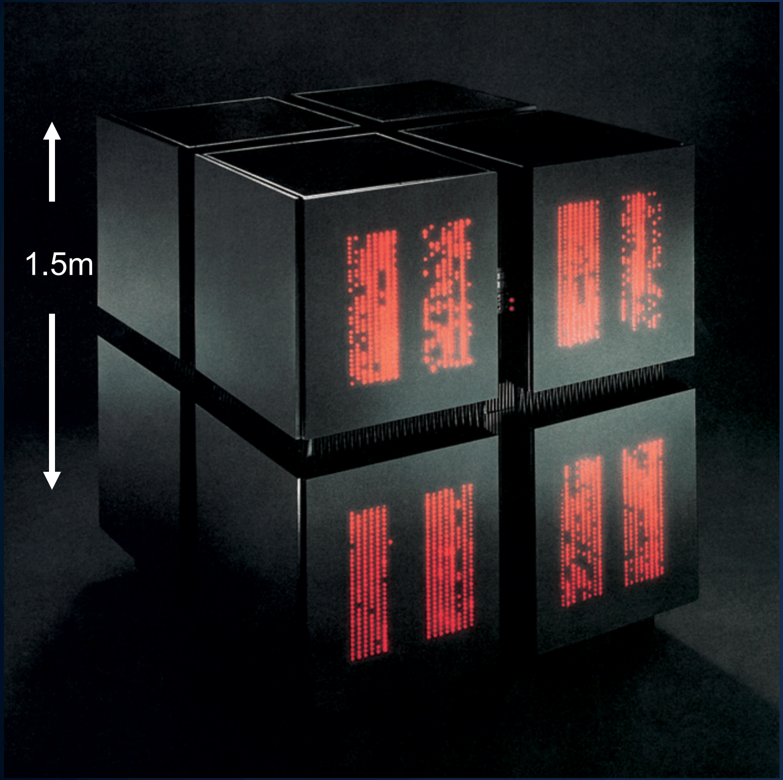
Flags are sometimes done with Boolean and switch points

F	F	T	T	T	F	F	T

Mapping Data Parallelism to Real Hardware

Connection Machine (CM-1,2)

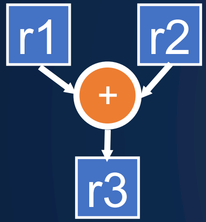
Because communication is more important than processors

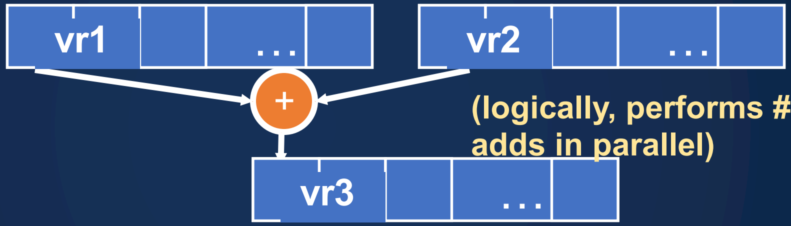


- Designed for AI by Thinking Machines Corporation (Hillis and Handler)
- CM-1 and CM-2 SIMD Design
  - 65,536 1-bit processors with 4 KB of memory each
  - 12-D boolean n-cube network (Feynman)
  - CM-2 add 1 floating point processor per 32 1-bit
- Programmed with data parallel languages
  - *Lisp
  - C*
- CM-5 was RISC+Vectors

SIMD/Vector Processorsr Use Data Parallelism

- SIMD instructions operate on a vector of elements
  - **These are specified as operations on vector registers**

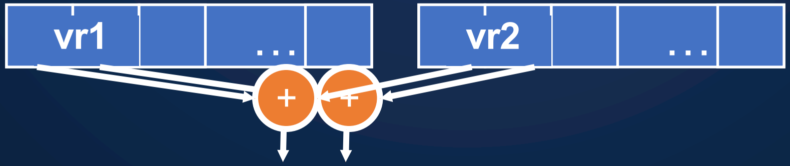




(logically, performs # elts adds in parallel)

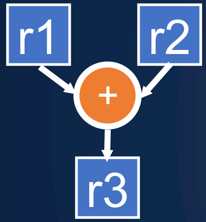
  - **Vectors "virtualize" the # of lanes (registers wider than #ALUs)**
  - **SIMD on CPUs does not)**

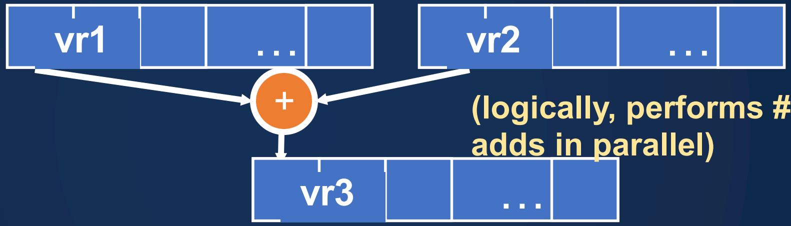
(performs #pipes adds in parallel)



SIMD/Vector Processorsr Use Data Parallelism

- SIMD instructions operate on a vector of elements
  - These are specified as operations on vector registers

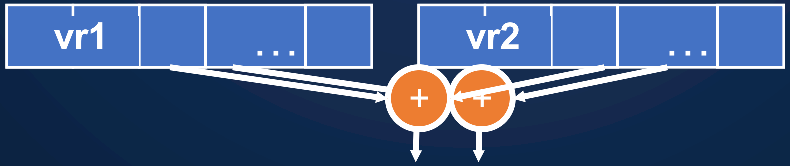




(logically, performs # elts adds in parallel)

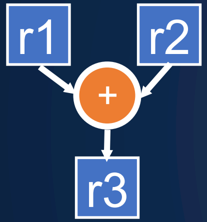
  - Vectors "virtualize" the # of lanes (registers wider than #ALUs)
  - SIMD on CPUs does not)

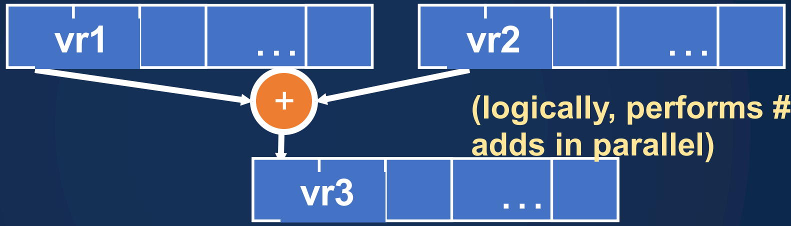
(performs #pipes adds in parallel)



SIMD/Vector Processorsr Use Data Parallelism

- SIMD instructions operate on a vector of elements
  - **These are specified as operations on vector registers**

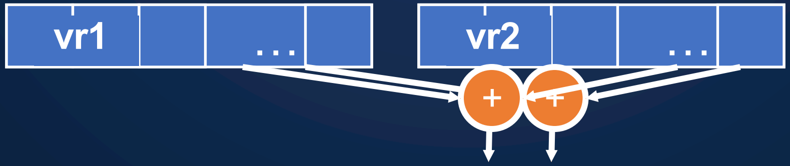




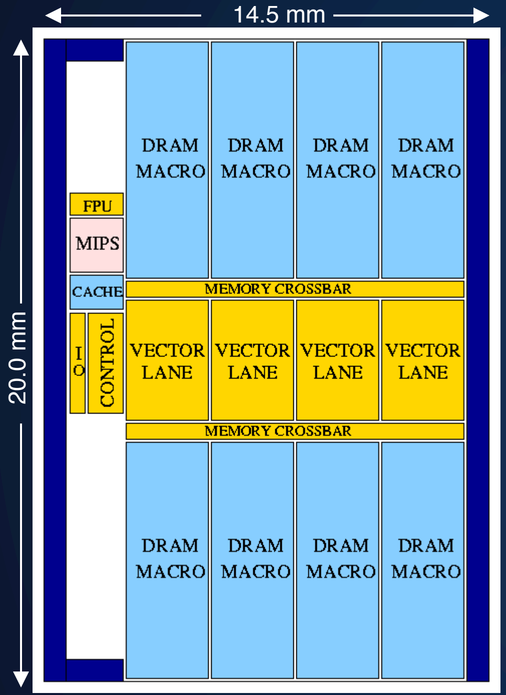
(logically, performs # elts adds in parallel)

  - **Vectors "virtualize" the # of lanes (registers wider than #ALUs)**
  - **SIMD on CPUs does not**

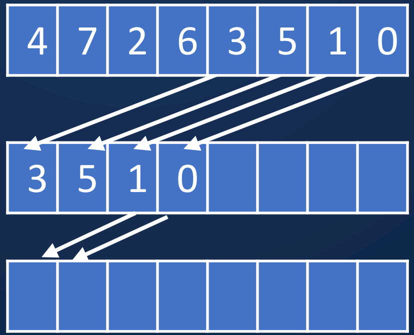
(performs #pipes adds in parallel)



VIRAM Processor at Berkeley

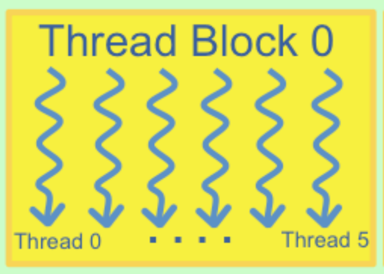


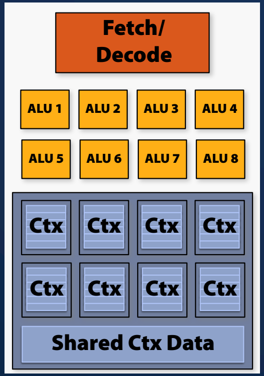
- MIPS Scalar core + 4-lane vector at 200 MHz (in 2002)
- 32-wide vector registers (in 64b)
- Peak vector performance
  - 1.6/3.2/6.4 Gops wo. multiply-add (64b/32b/16b operations)
  - 1.6 Gflops (single-precision)
- Transistor count: ~130M
- Reduction scan (in-register permutation instructions)

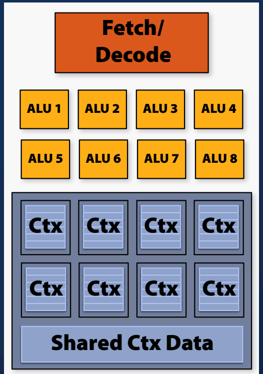


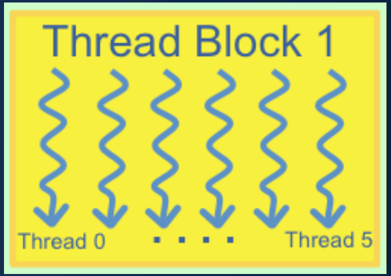
Mapping to GPUs

- For n-way parallelism may use n threads, divided into blocks
- Merge across statements (so A=B; C=A; is a single kernel)
- Mapping threads to ALUs and blocks to SMs is compiler / hardware problem









Bottom Line

- Branches are still expensive on GPUs
- May pad with zeros / nulls etc. to get length
- Often write code with a guard (if i < n), which will turn into mask – fine if n is large
- Non-contiguous memory is supported, but will still have a higher cost
- Enough parallelism to keep ALUs busy

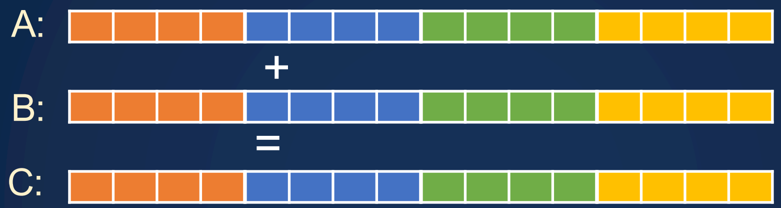
and hide latency, memory/scheduling tradeoff

Mapping Data Parallelism to SMPs (and MPPs)

n-way parallelism onto p-way hardware

- Binary and unary operations

C = A+B



Cost O(n/p)

p speedup

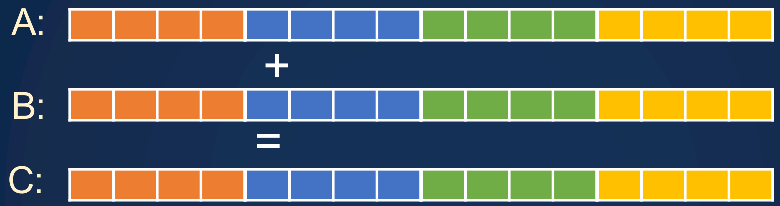
- If arrays are not "aligned" then false sharing / communication require

Mapping Data Parallelism to SMPs (and MPPs)

n-way parallelism onto p-way hardware

- Binary and unary operations

C = A+B

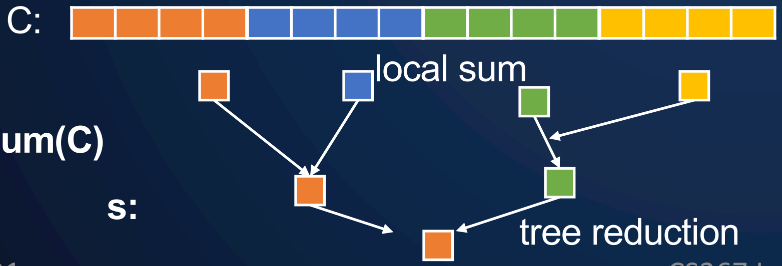


Cost O(n/p)

p speedup

- If arrays are not "aligned" then false sharing / communication require
- Reductions and broadcasts

s = sum(C)



Cost O(n/p

+ log p)

Almost p speedup

Parallel prefix cost on p "big" processors

Time for this algorithm in parallel:

- $T_p(n) = O(n/p + \log p)$

serial time on each processor

communication and computation up and down the processor tree

Compute local prefix sums in n/p steps

Updates across processors in log p steps

The myth of log n

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

- Say n = k*p (k = 1,000,000 elements per proc)
  – Cost = (k adds) + ($\log_2 P$ steps) + (k adds)

compute and store k values a[0]..a[k-1]

parallel scan on a[k-1] values

add 'my' scan result to a[0]..a[k-1]

(2,000,000 local adds are serial for each processor, of course)

Key to implementing data parallel algorithms on clusters, SMPs, MPPs, i.e., modern supercomputers

Summary of Data Parallelism

- Sequential semantics (or nearly) is very nice
  - Debugging is much easier without non-determinism
  - Correctness easier to reason about
- Cost model is independent of number of processors
  - How much inherent parallelism
- Need to “throttle” parallelism
  - n >> p can be hard to map, especially with nesting
  - Memory use is a problem
- More reading
  - Classic paper by Hillis and Steele “Data Parallel Algorithms” https://doi.org/10.1145/7902.7903 and on Youtube
  - Blelloch the NESL languages and “NESL Revisited paper, 2006