CS 267: Introduction to Data Parallelism Lecture 7

Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spr2019/

Lessons from Today's Lecture

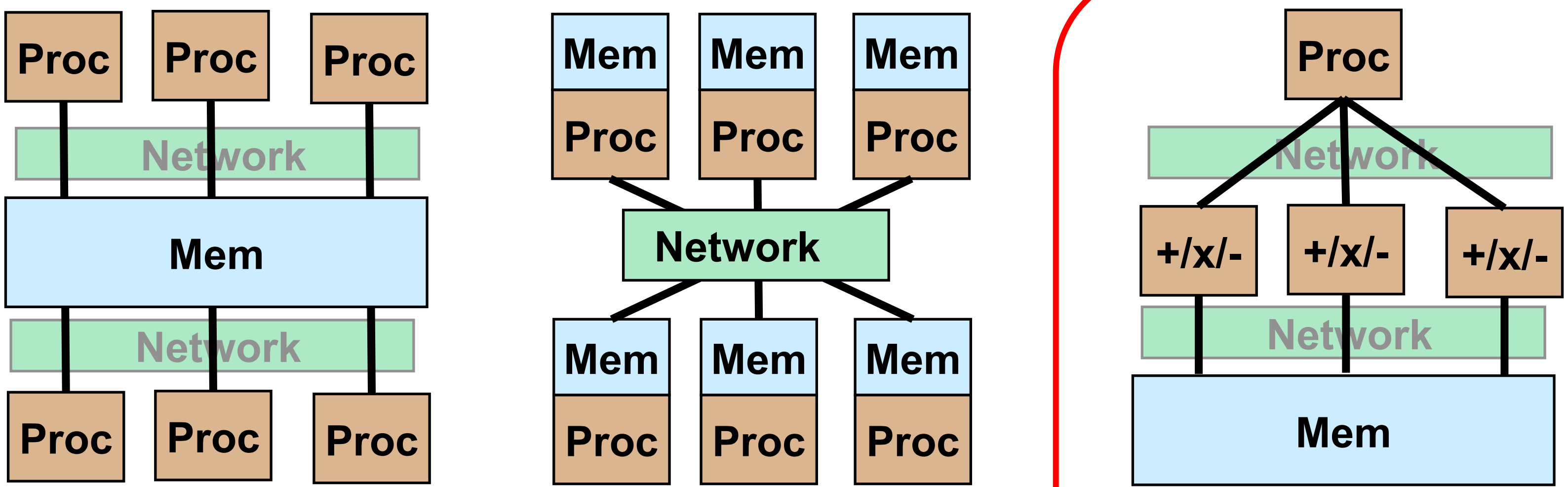
- Data parallelism is beautiful!
- Automatically mapping it to (today's) hardware is hard

- Many parallel programming models use some data parallel features
  - GPUs
  - MPI collectives
  - Cloud MapReduce

- Surprising things you can do with scans
- Useful in designing (nontrivial) parallel algorithms

"Every nontrivial parallel algorithm uses a prefix scan"

Parallel Machines and Programming



	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
Ideal cost	Cost of a read/write is constant	Message time depends on size, but not location	Assume unbounded # of arithmetic units

- These are the natural "abstract" machine models

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

- applied to each pair

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

Data Parallel Programming: Broadcast

- Broadcast fill a value into all elements of an array

a = scalar
B = a

a: 3

broadcast

B:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

- Useful for a*X+Y called axpy, saxpy, daxpy

a*X + Y

a: 2

X:	3	1	1	2	3	3	4	2	2	2	1	3	1	1	1	3	3	2	1
Y:	1	2	0	2	1	3	1	0	2	2	1	3	0	1	1	0	3	0	1

axpy

Z:	7	4	2	6	7	9	9	4	6	6	3	9	2	3	3	6	9	4	3

Memory Operations: Strided and Scatter / Gather

- Array assignment work 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]
C: double [0:4, 0:4]
X: int [0:4] = [3, 0, 4, 2, 1]

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)
A = C [*,3]      // copy column of C
```

- Gather (indexed) values from one array

```
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]
```

Questions?
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

- Related: Segmented scans to be presented later

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

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

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]

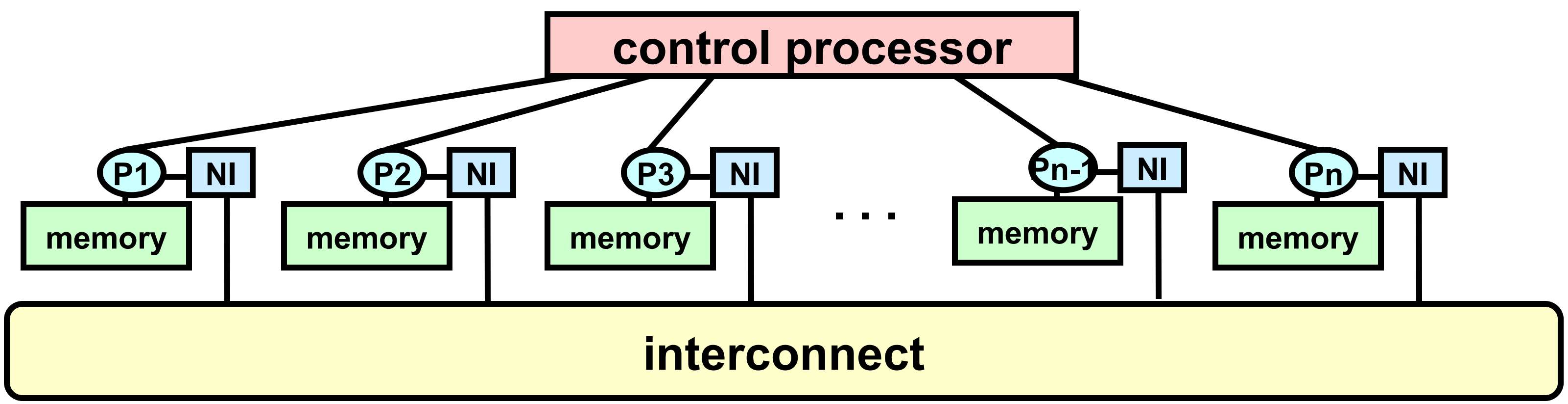
Note that inclusive version from the exclusive by applying the operation across vectors: scan_inclusive(X) = X $\circledcirc$ scan_exclusive(X).

To go the other way you need an inverse for $\circledcirc$ **(- for +)**
You can convert both directions using vector shifts left or right.

Idealized Hardware and Performance Model

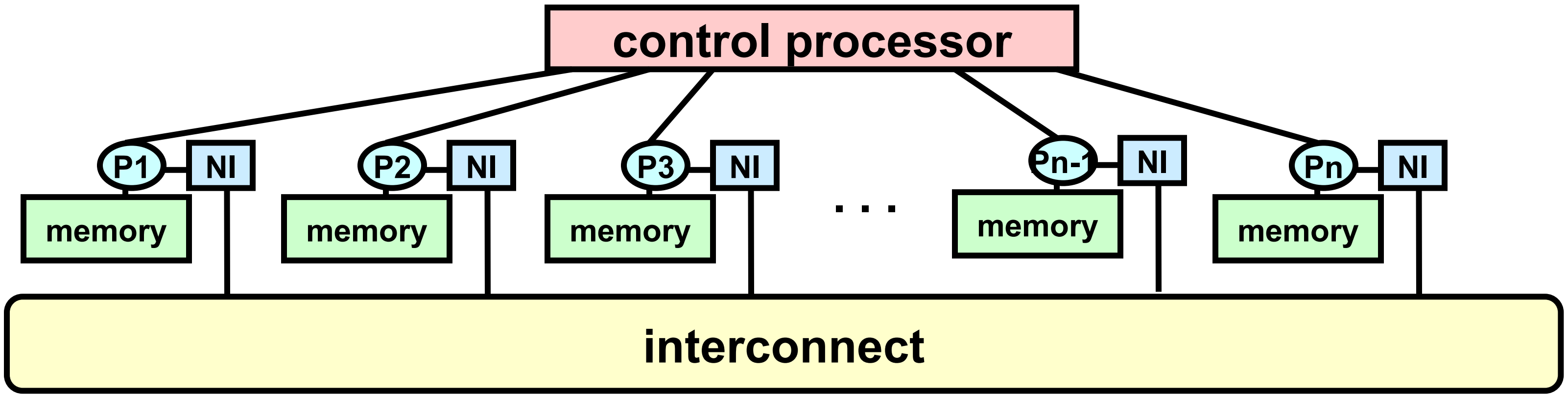
SIMD Systems Implemented Data Parallelism

- 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.**
- Originally machines were specialized to scientific computing, few made (CM2, Maspar)



Ideal Cost Model for Data Parallelism

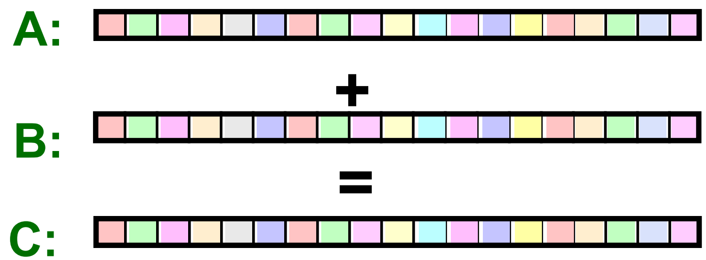
- Machine
  - An unbounded number of processors (p)
  - Control overhead is free
  - Communication is free
- Shows the inherent parallelism (inherent serialization)
- Called the algorithm's "span"
- Defines a lower bound on real machines



Cost on Ideal Machine (Span)

- Span for unary or binary operations (pleasingly parallel)

C = A+B

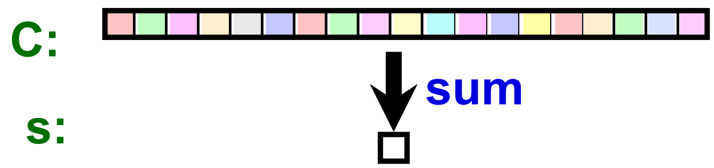


Cost O(1)
since p is unbounded

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

- Reductions and broadcasts

s = sum(C)

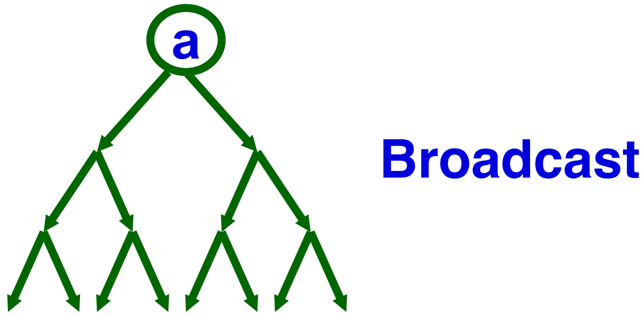


Cost O(log(n))

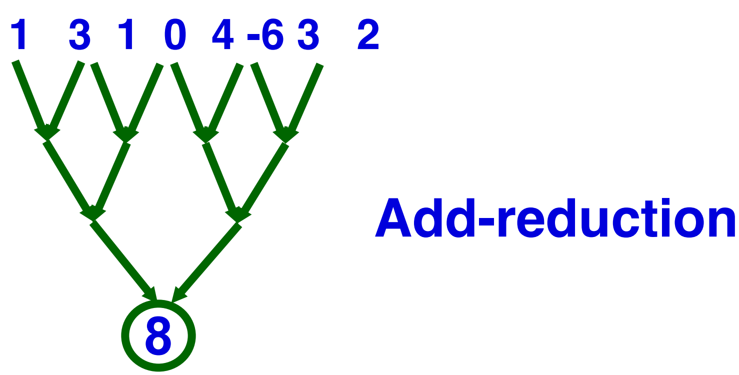
Using a tree of processors

Broadcast and reduction use processor trees

- 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.



Important of Associativity

- Is it “OK” to do a reduction on floating point values?
- Neither + nor * are associative in floating point arithmetic due to rounding

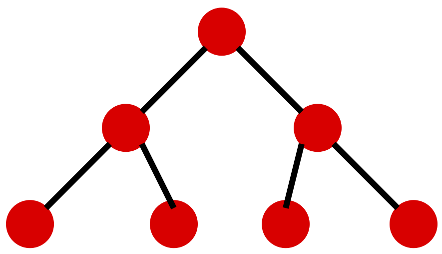
$$(1+10^{20})+ -10^{20} \neq 1+ (10^{20} + -10^{20})$$

- Answer: You won’t get the same answer, and may get different ones (up to associativity) on different machines
- Often OK, i.e., floating point +/* treated as associative

- MPI (coming soon) assumes associativity in reductions
- SPARK assumes associativity and commutativity

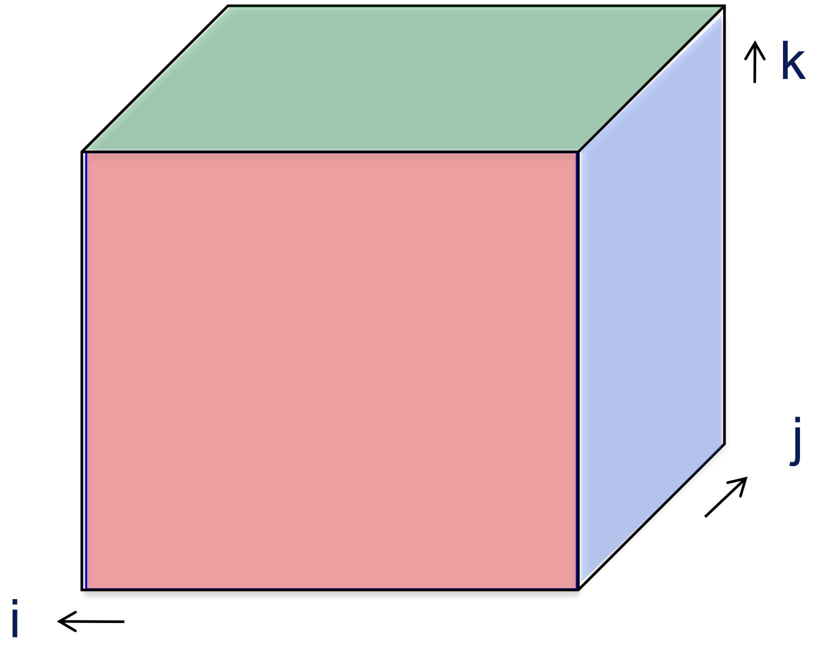
Can reductions go faster: No, log n lower bound on any function of n variables!

- 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
- Use induction to show that 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



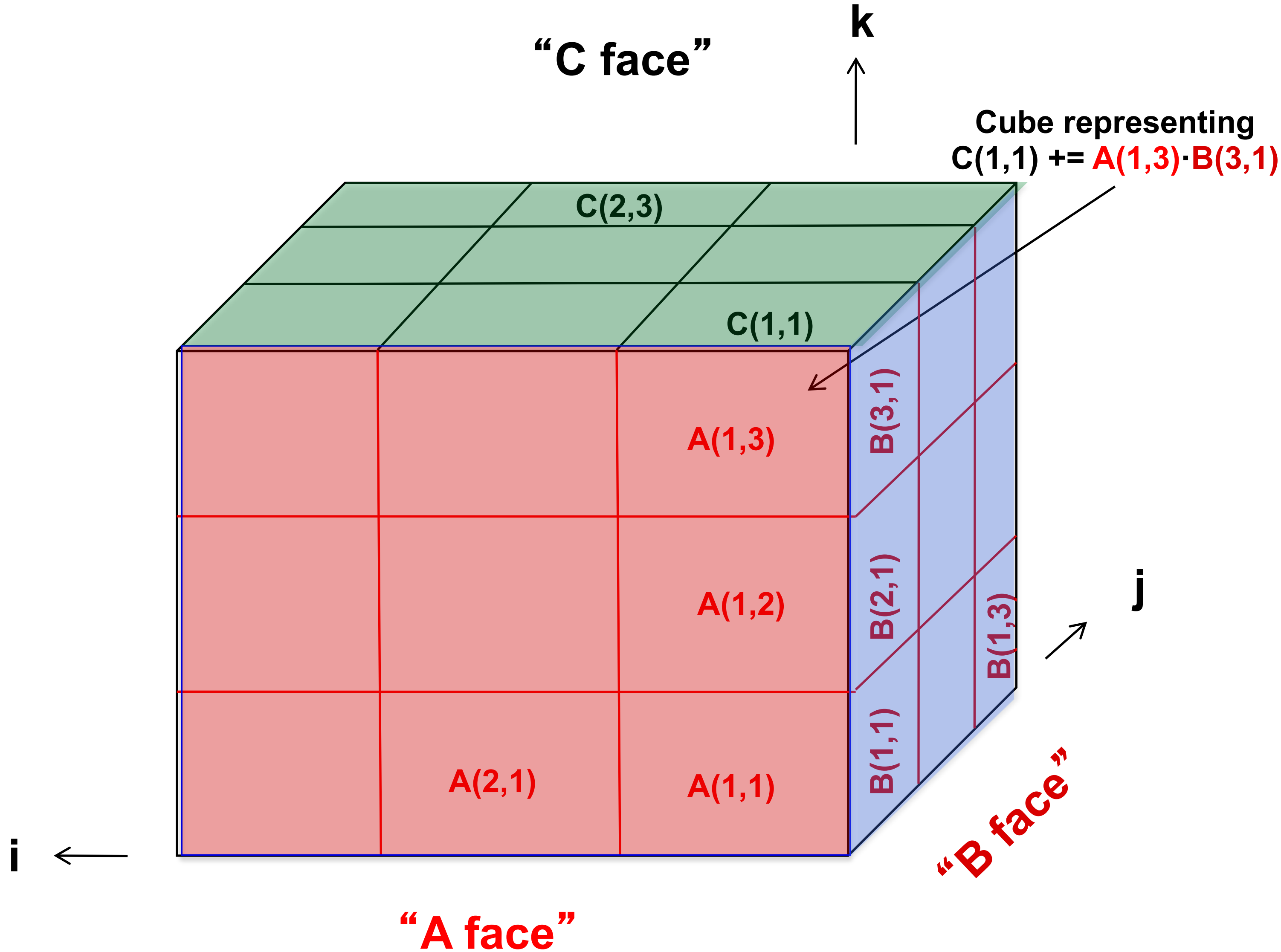
Multiplying n-by-n matrices in O(log n) time

- 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

Related to Communication-Optimal "2.5D" MatMul



- Processors execute internal sub-cubes

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

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

- 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
- 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)

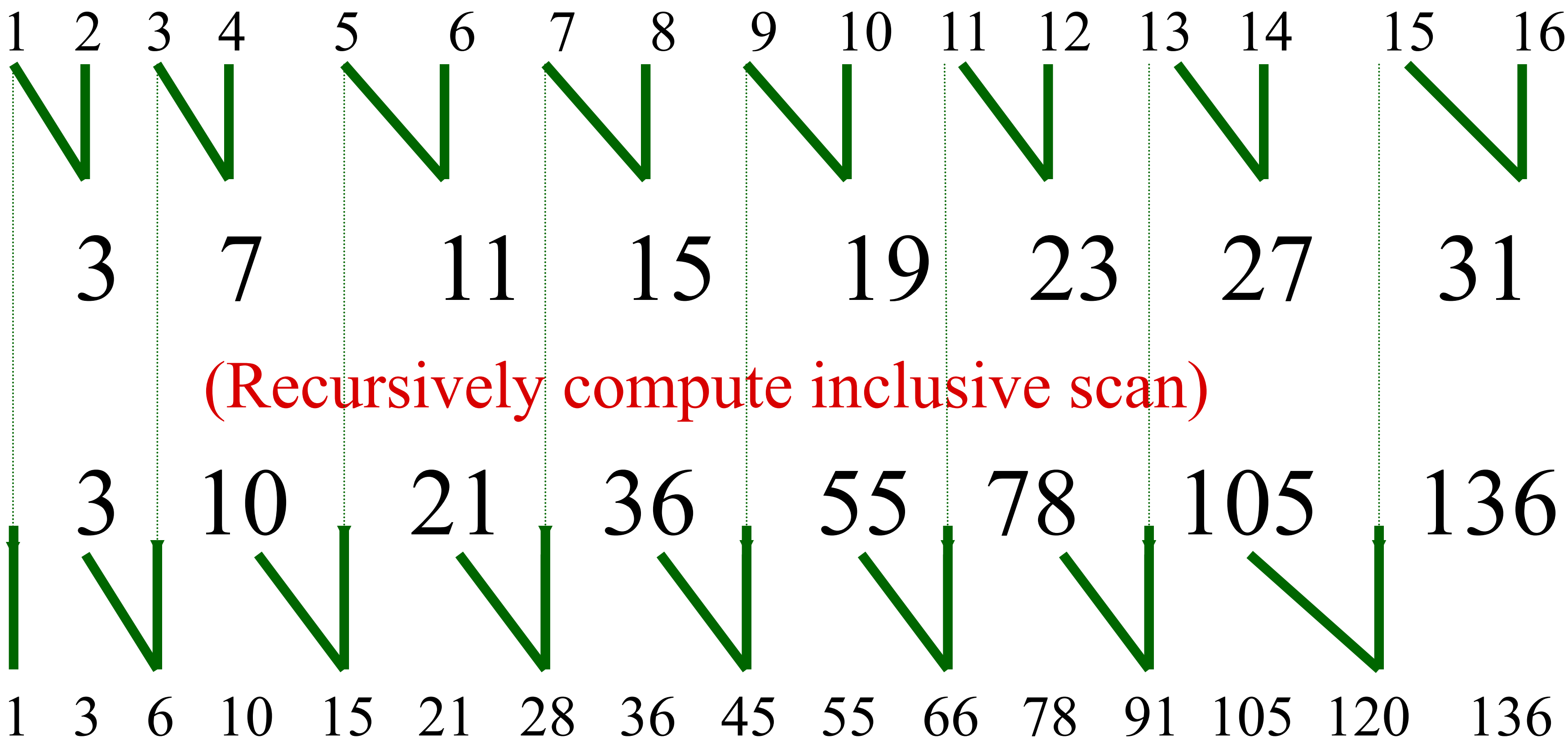
Is there any value in adding, say, 5+6+7+8?

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

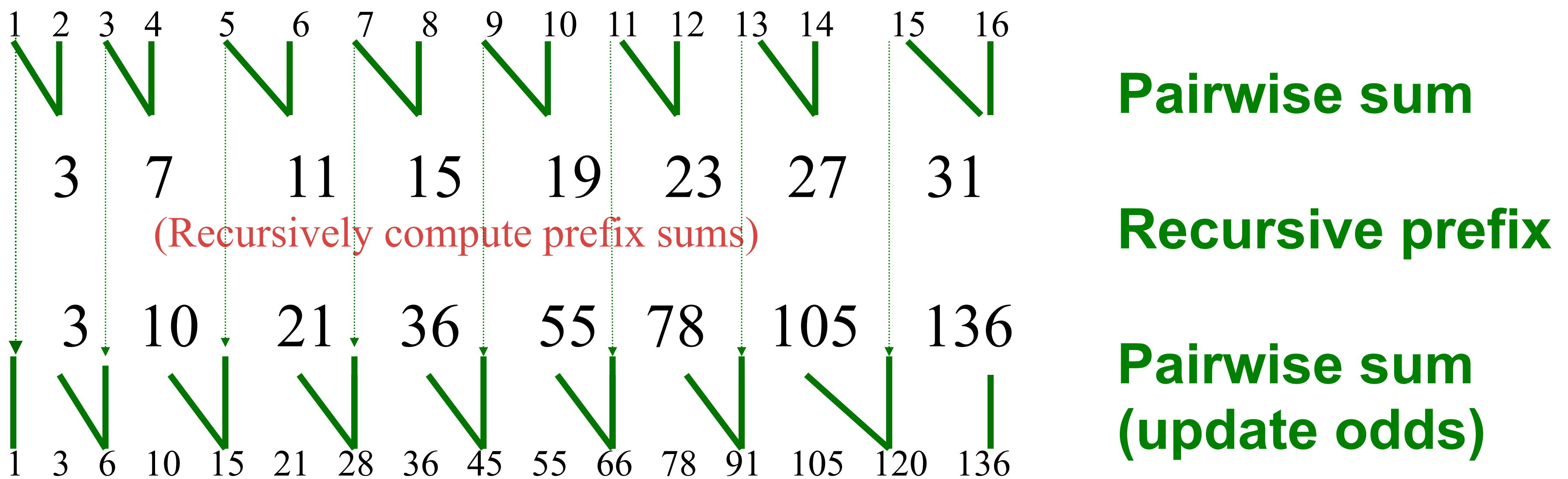
Suppose we added 1+2, 3+4, etc. pairwise -- what could we do?

Sum Scan (aka prefix sum) in parallel

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



Scan (parallel prefix) cost



Time for this algorithm on one processor (work)

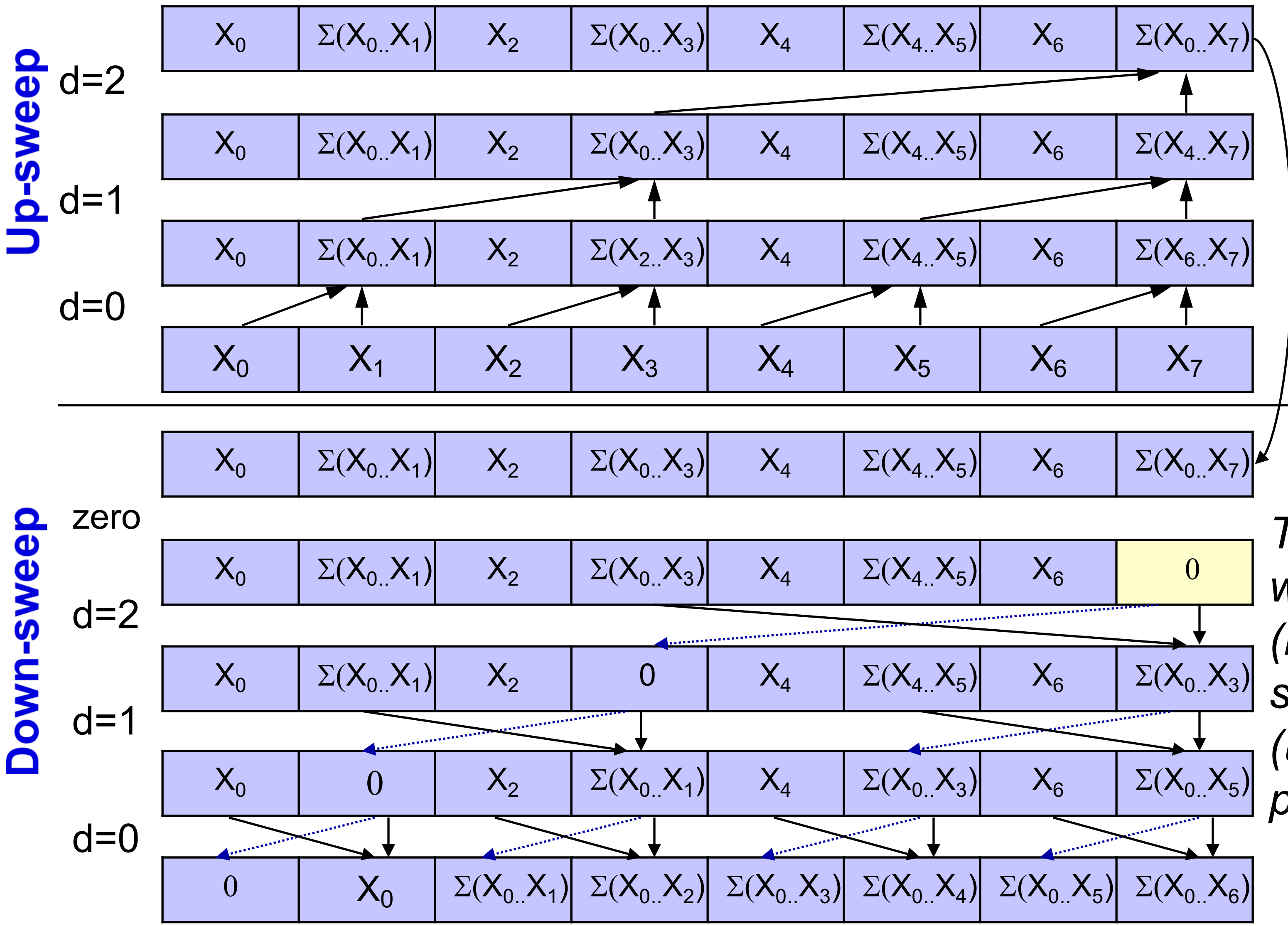
- $T_1(n) = n/2 + n/2 + T_1(n/2) = n + T_1(n/2) = 2n - 1$

Time on unbounded number of processors (span)

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

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

Non-recursive view of parallel prefix scan



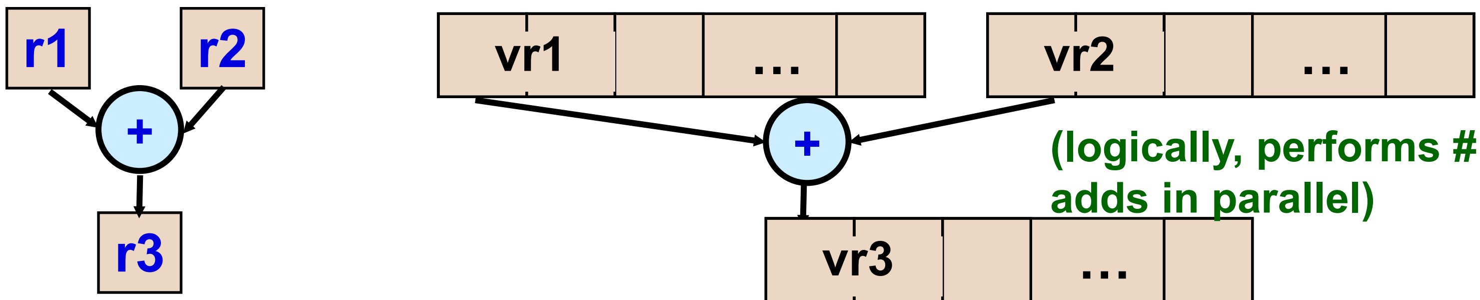
This is both work-efficient (n adds) and space-efficient (update in place)

Algorithm due to Blelloch

Real Hardware (Today)

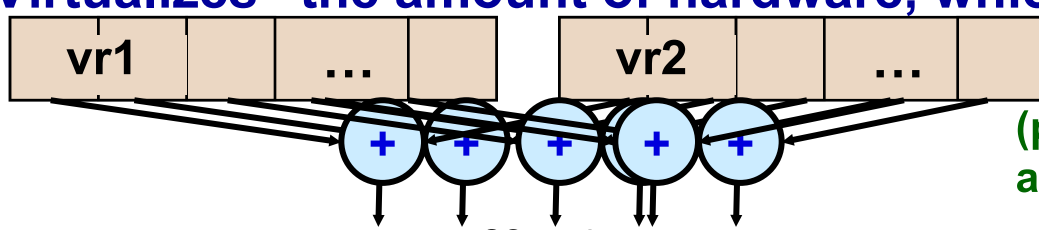
Vector Machines Use Data Parallelism

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



(logically, performs # elts adds in parallel)

- Old supercomputer vector register: ~32-64 elts
  - The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4
- The hardware performs a full vector operation in
  - #elements-per-vector-register / #pipes steps
  - E.g., 64 elements in register, but only 8 fp adders to use
  - "Virtualizes" the amount of hardware, which is n

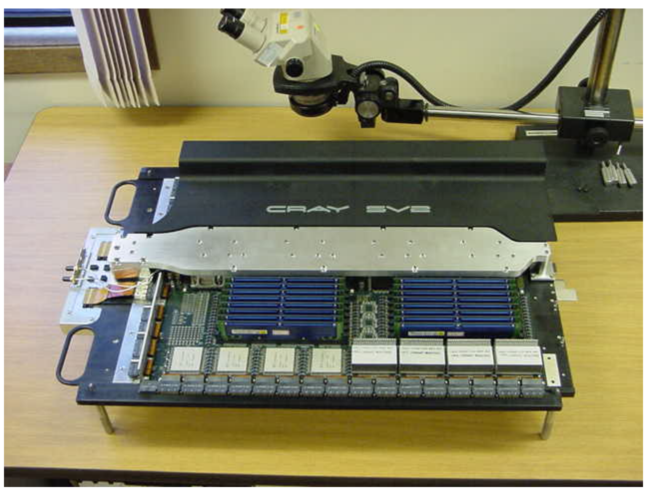


(performs #pipes adds in parallel)

Cray X1: *Parallel Vector Architecture*

Cray combined several technologies in the X1

- **12.8 Gflop/s Vector processors (MSP)**
- **Shared caches (unusual on earlier vector machines)**
- **4 processor nodes sharing up to 64 GB of memory**
- **Single System Image to 4096 Processors**
- **Remote put/get between nodes (faster than MPI)**

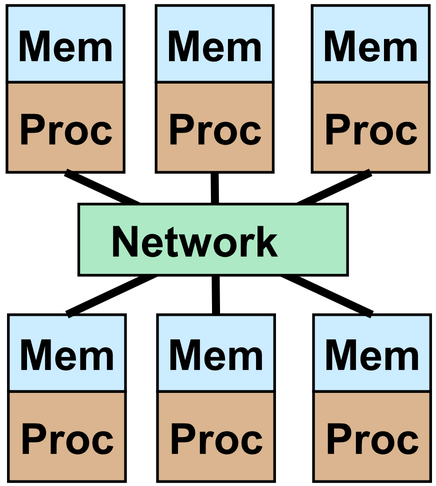


- **Expensive to design and build, market too small**

SIMD instructions use Data Parallelism

- **SIMD instructions on microprocessors are vectors**
  - Shorter than old vector supercomputers (e.g., 256 bits)
  - They don't virtualize the hardware (arithmetic units), so each processor version may require code rewrites
- **Reductions and broadcasts are in register**
  - Require inside-register data movement
- **Assuming vector length (or SIMD width) are small constants → no theoretical speedup**
  - But in practice this can make a big different (2-16x…)
  - And algorithms may still be useful
- **Revisit these ideas with GPUs**

Data parallelism on Distributed Memory



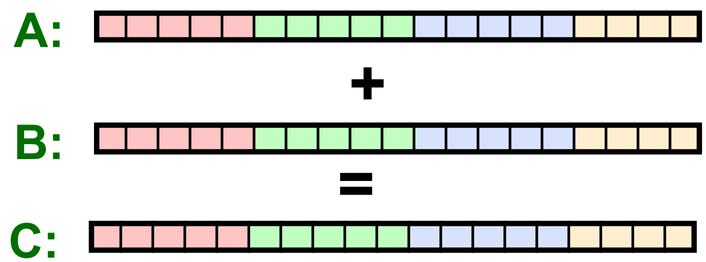
Distributed Memory
Processors execute own instruction stream
Communicate by sending messages
Message time depends on size, but not location

- **Today's parallel machines**
  - **Powerful processors**
  - **Distributed memory (at scale)**
  - **Clusters or MPPs (Massively Parallel Processors)**
- **Need to map n-way parallelism to p-way**
  - **Attempts to do this automatically**
- **High Performance Fortran**
  - **Large effort in the 90s**
  - **Semi-automatic: Data layout hints were necessary**
  - **And it was still hard**
- **But still useful manually**

Mapping Data Parallelism to Clusters

- Binary and unary operations on MPPs

C = A+B

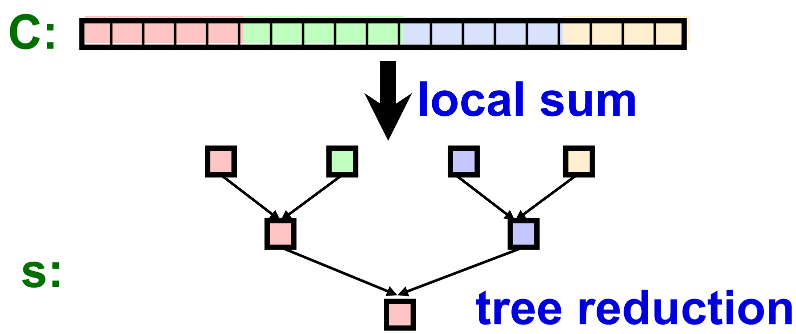


Cost O(n/p)

p speedup

- If arrays are not “aligned” then communication required
- Reductions and broadcasts

s = sum(C)

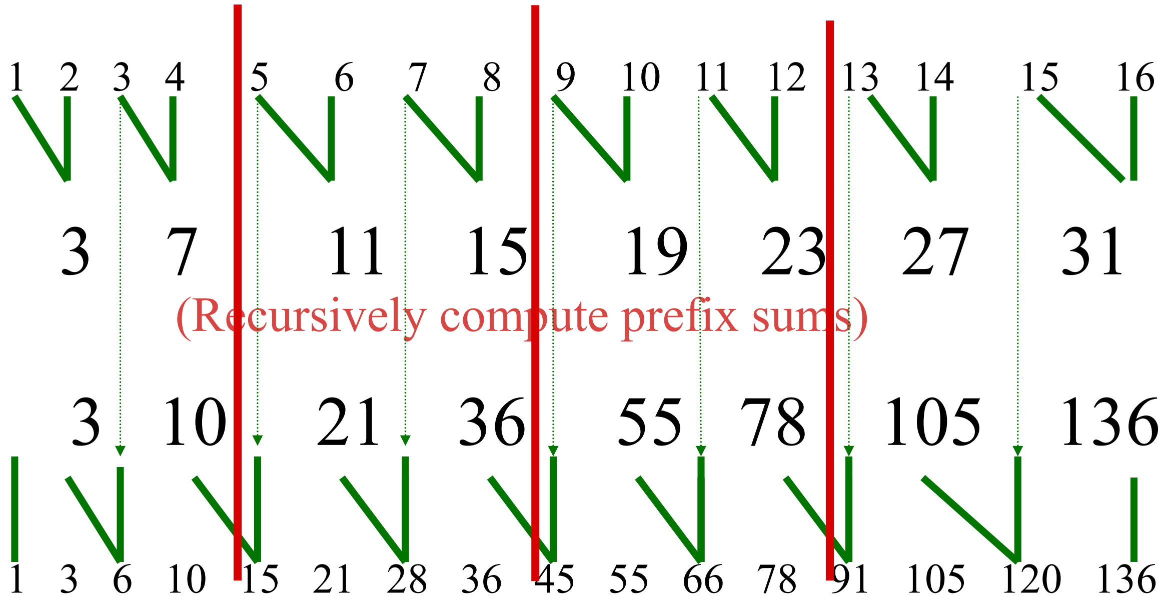


Cost O(n/p

+ log p)

Almost p speedup

Parallel prefix cost on p processors



Compute local prefix sums in n/p steps

Updates across processors in log p steps

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

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 = $\boxed{\text{(k adds)}}$ + $\boxed{(\log_2 P \text{ steps})}$ + $\boxed{\text{(k adds)}}$

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

Data Parallelism is an Elegant Programming Model

- Strict data parallelism has serial semantics:
  - E.g., no difference from executing A+B one element at a time or in parallel
- Reductions also preserve serial semantics for truly associative operations:
  - \+ * min, etc. on integers and more;
  - some differences for floating point due to order of evaluation (but can be deterministic, i.e., the same result every time)
- Easy to understand and reason about
- “In spirit” in MPI collectives, CUDA, MapReduce...

Limitations:

- Some algorithms (e.g., adaptive) don’t fit easily
- Non-trivial to implement on some hardware

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

- 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

3	2	4	1	5	3	3	1

3	4	1	3	1

Application: Radix Sort (serial algorithm to start)

4	7	2	6	3	5	1	0

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

4	2	6	0	7	3	5	1

XX0 < XX1 (evens before odds)

Bit_0=0 **Bit_0=1**

4	2	6	0	7	3	5	1

Stably sort entire array on next bit

4	0	5	1	2	6	7	3

X0X < X1X

Bit_1=0 **Bit_1=1**

4	0	5	1	2	6	7	3

Stably sort on next bit

0	1	2	3	4	5	6	7

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

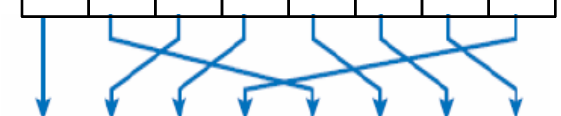
Bit_2=0 **Bit_2=1**

A “stable” sort means it preserves the ordering. unless they have to switch based on the current bit

Application: 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	epos = 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
0-0+4 =4	1-1+4 =4	2-1+4 =5	3-2+4 =5	4-3+4 = 5	5-3+4 =6	6-3+4 =7	7-4+4 =7		
4	4	5	5	5	6	7	7		oddpos = indx + totalEvens – epos
0	4	1	2	5	6	7	3		pos = if evens then esum else oddpos
4	7	2	6	3	5	1	0		

Using two masked assignments



4	2	6	0	7	3	5	1

Scatter input using pos as index

Repeat with next bit to left until done

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

  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

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

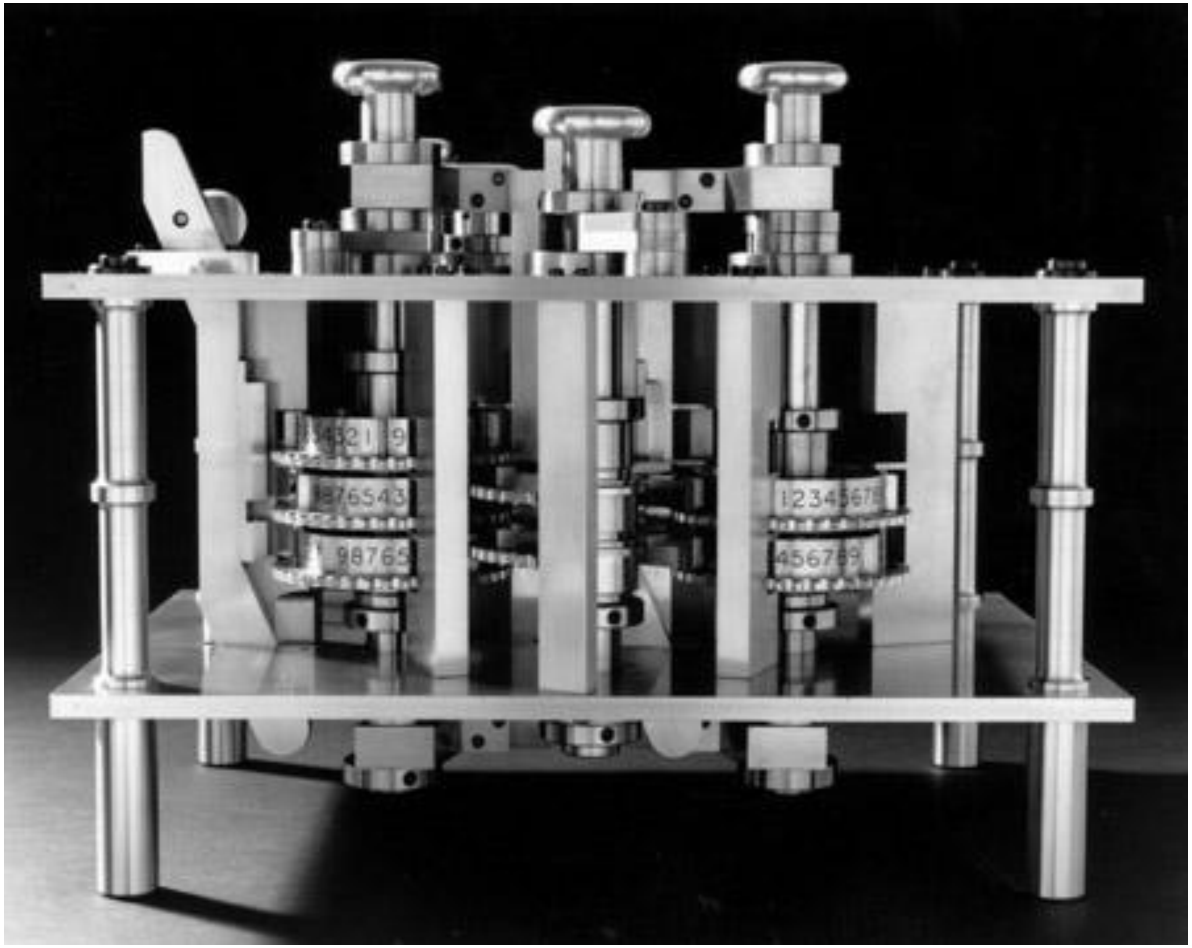
$$= M[i] * M[i-1] * \ldots M[0] * \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

… 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

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

Flags are sometimes done with Boolean and switch points

F	F	T	T	T	F	F	T

Result

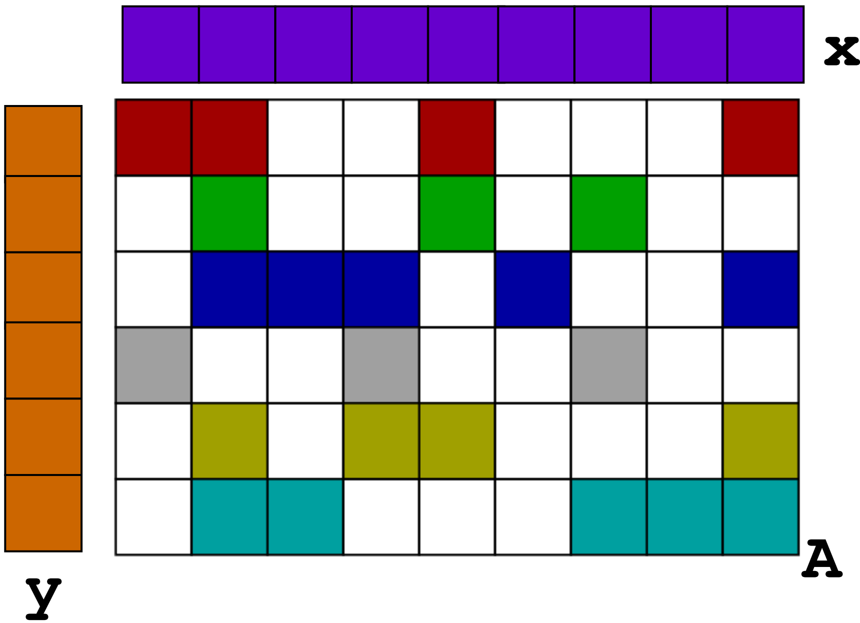
1	3	3	7	12	6	13	8

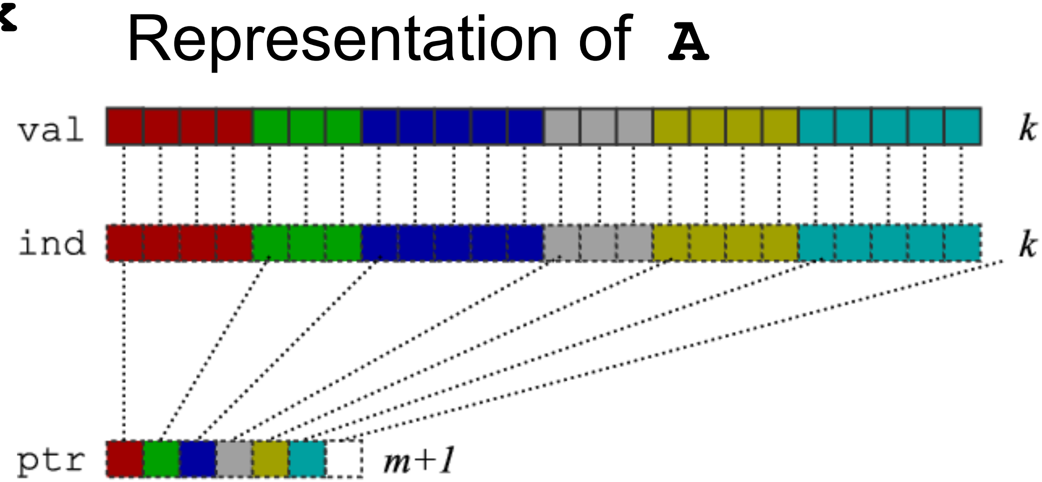
SpMV in Compressed Sparse Row (CSR) Format

SpMV: y = y + A*x

Sparse matrices: only store, do arithmetic, on nonzero entries

CSR format is simplest one of many possible data structures for A





Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j) \times x(j)$

```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
      y[i] = y[i] + val[k]*x[ind[k]]
```

SPMV (Segmented Suffix Scan)

- Sparse Matrix-Vector Multiplication (SPMV)

X=

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

A:

PTR =

0	2	5	7	9

Row Starts

IND =

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

VAL =

1	1	1	2	2	2	1	2	2	1	3	1

```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
SUMS = SUM_SUFFIXSCAN_SEGD_INCL (PROD,FLAG)
Y = SUMS(PTR)
```

FLAG =	1	0	1	0	0	1	0	1	0	1	0	0
PROD =	2	1	1	4	2	4	1	4	2	1	6	1
SUMS =	**3**	1	**7**	6	2	**5**	1	**6**	2	**8**	7	1

(SUFFIXSCAN goes right to left)

Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Blelloch, Heroux, Zagha. CMU-CS-93-173

Application: 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

Slide source: Alan Edelman

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

Lessons from Data Parallel Languages

- 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

See: Blelloch "NESL Revisited", Intel Workshop 2006