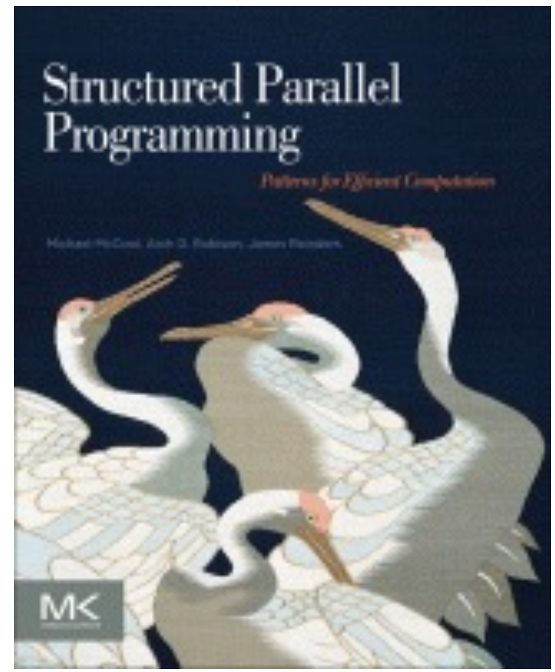
Parallel Functional Programming
Data Parallelism

Mary Sheeran

Data parallelism is the key to achieving scalability. Merely dividing up the source code into tasks using functional decomposition will not give more than a constant factor speedup. To continue to scale to ever larger numbers of cores, it is crucial to generate more parallelism as the problem grows larger. Data parallelism achieves this.



Structured Parallel Programming : Patterns for Efficient Computation
Michael McCool, James Reinders, and Arch Robison

Interesting book from Elsevier, 2012, available free online at Chalmers library (just search)
Ideas from this lecture are very prominent in it

Data parallelism

Introduce data structures along with parallel operations on them

Often data parallel arrays

Canonical example : NESL (NESted-parallel Language)

(Blelloch)

Data parallelism

Introduce data structures along with parallel operations on them

Often data parallel arrays

Canonical example : NESL (NESted-parallel Language) (Blelloch)

See video of Blelloch's ICFP10 invited talk in lecture description

NESL

concise (good for specification, prototyping)

allows programming in familiar style (but still gives parallelism)

allows nested parallelism (as distinct from flat)

associated language-based cost model

gave decent speedups on wide-vector parallel machines of the day

Hugely influential!

http://www.cs.cmu.edu/~scandal/nesl.html

NESL

Parallelism without concurrency!

Completely deterministic (modulo floating point noise)

No threads, processes, locks, channels, messages, monitors, barriers, or even futures, at source level

Based on Blelloch's thesis work: [Vector Models for Data-Parallel Computing, MIT Press 1990]

NESL

Parallelism without concurrency!

Completely deterministic (modulo floating point noise)

No threads, processes, locks, channels, messages, monitors, barriers, or even futures, at source level

Based on Blelloch's thesis work: [Vector Models for Data-Parallel Computing, MIT Press 1990]

NESL

NESL is a sugared typed lambda calculus with a set of array primitives and an explicit parallel map over arrays

To be useful for analyzing parallel algorithms, NESL was designed with rules for calculating the work (the total number of operations executed) and depth (the longest chain of sequential dependence) of a computation.

NESL

For modeling the cost of NESL we augment a standard call by value operational semantics to return two cost measures: a DAG representing the sequential dependences in the computation and a measure of the space taken by a sequential implementation. We show that a NESL program with w work (nodes in the DAG) d depth (levels in the DAG) and s sequential space can be implemented on a p processor butterfly network, hypercube or CRCW PRAM using O(w/p + d log p) time and O (s + dp log p) reachable space. For programs with sufficient parallelism these bounds are optimal in that they give linear speedup and use space within a constant factor of the sequential space.

Quotes are from ICFP'96 paper

A Provable Time and Space Efficient Implementation of NESL

Guy E. Blelloch and John Greiner
Carnegie Mellon University
{blelloch,jdg}@cs.cmu.edu

Abstract

In this paper we prove time and space bounds for the implementation of the programming language NESL on various parallel machine models. NESL is a sugared typed λ-calculus with a set of array primitives and an explicit parallel map over arrays. Our results extend previous work on provable implementation bounds for functional languages by considering space and by including arrays. For modeling the cost of NESL we augment a standard call-by-value operational semantics to return two cost measures: a DAG representing the sequential dependences in the computation, and a measure of the space taken by a sequential implementation. We show that a NESL program with w work (nodes in the DAG), d depth (levels in the DAG), and s sequential space can be implemented on a p processor butterfly network, hypercube, or CRCW PRAM using $O(w/p + d \log p)$ time and $O(s + dp \log p)$ reachable space.[1] For programs with sufficient parallelism these bounds are optimal in that they give linear speedup and use space within a constant factor of the sequential space.

The idea of a provably efficient implementation is to add to the semantics of the language an accounting of costs, and then to prove a mapping of these costs into running time and/or space of the implementation on concrete machine models (or possibly to costs in other languages). The motivation is to assure that the costs of a program are well defined and to make guarantees about the performance of the implementation. In previous work we have studied provably time efficient parallel implementations of the λ-calculus using both call-by-value [3] and speculative parallelism [18]. These results accounted for work and depth of a computation using a profiling semantics [29, 30] and then related work and depth to running time on various machine models.

This paper applies these ideas to the language NESL and extends the work in two ways. First, it includes sequences (arrays) as a primitive data type and accounts for them in both the cost semantics and the implementation. This is motivated by the fact that arrays cannot be simulated efficiently in the λ-calculus without arrays (the simulation of an array of length n using recursive types requires a $\Omega(\log n)$ slowdown). Second, it augments the profiling semantics with

Quotes

This paper adds the accounting of costs to the semantics of the language and proves a mapping of those costs into running time / space on concrete machine models

A Provable Time and Space Efficient Implementation of NESL

Guy E. Blelloch and John Greiner
Carnegie Mellon University
{blelloch,jdg}@cs.cmu.edu

Abstract

In this paper we prove time and space bounds for the implementation of the programming language NESL on various parallel machine models. NESL is a sugared typed λ-calculus with a set of array primitives and an explicit parallel map over arrays. Our results extend previous work on provable implementation bounds for functional languages by considering space and by including arrays. For modeling the cost of NESL we augment a standard call-by-value operational semantics to return two cost measures: a DAG representing the sequential dependences in the computation, and a measure of the space taken by a sequential implementation. We show that a NESL program with w work (nodes in the DAG), d depth (levels in the DAG), and s sequential space can be implemented on a p processor butterfly network, hypercube, or CRCW PRAM using $O(w/p + d \log p)$ time and $O(s + dp \log p)$ reachable space.[1] For programs with sufficient parallelism these bounds are optimal in that they give linear speedup and use space within a constant factor of the sequential space.

The idea of a provably efficient implementation is to add to the semantics of the language an accounting of costs, and then to prove a mapping of these costs into running time and/or space of the implementation on concrete machine models (or possibly to costs in other languages). The motivation is to assure that the costs of a program are well defined and to make guarantees about the performance of the implementation. In previous work we have studied provably time efficient parallel implementations of the λ-calculus using both call-by-value [3] and speculative parallelism [18]. These results accounted for work and depth of a computation using a profiling semantics [29, 30] and then related work and depth to running time on various machine models.

This paper applies these ideas to the language NESL and extends the work in two ways. First, it includes sequences (arrays) as a primitive data type and accounts for them in both the cost semantics and the implementation. This is motivated by the fact that arrays cannot be simulated efficiently in the λ-calculus without arrays (the simulation of an array of length n using recursive types requires a $\Omega(\log n)$ slowdown). Second, it augments the profiling semantics with

Connection Machine

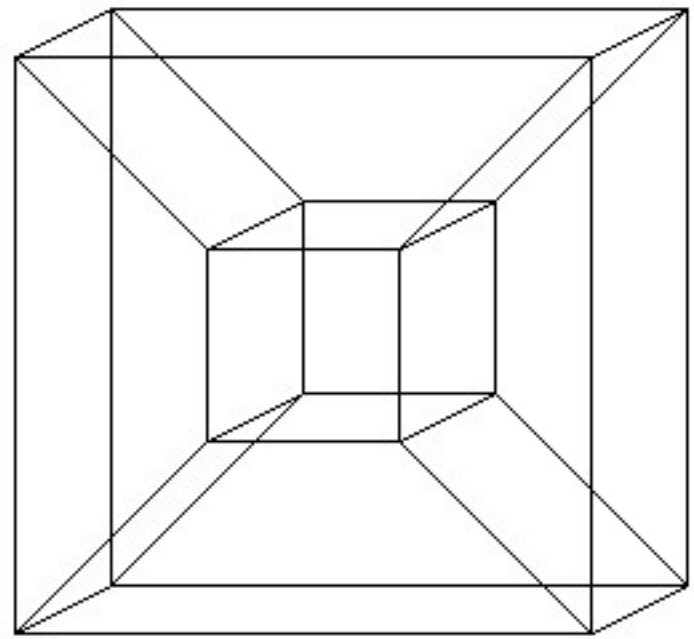
First commercial massively parallel machine

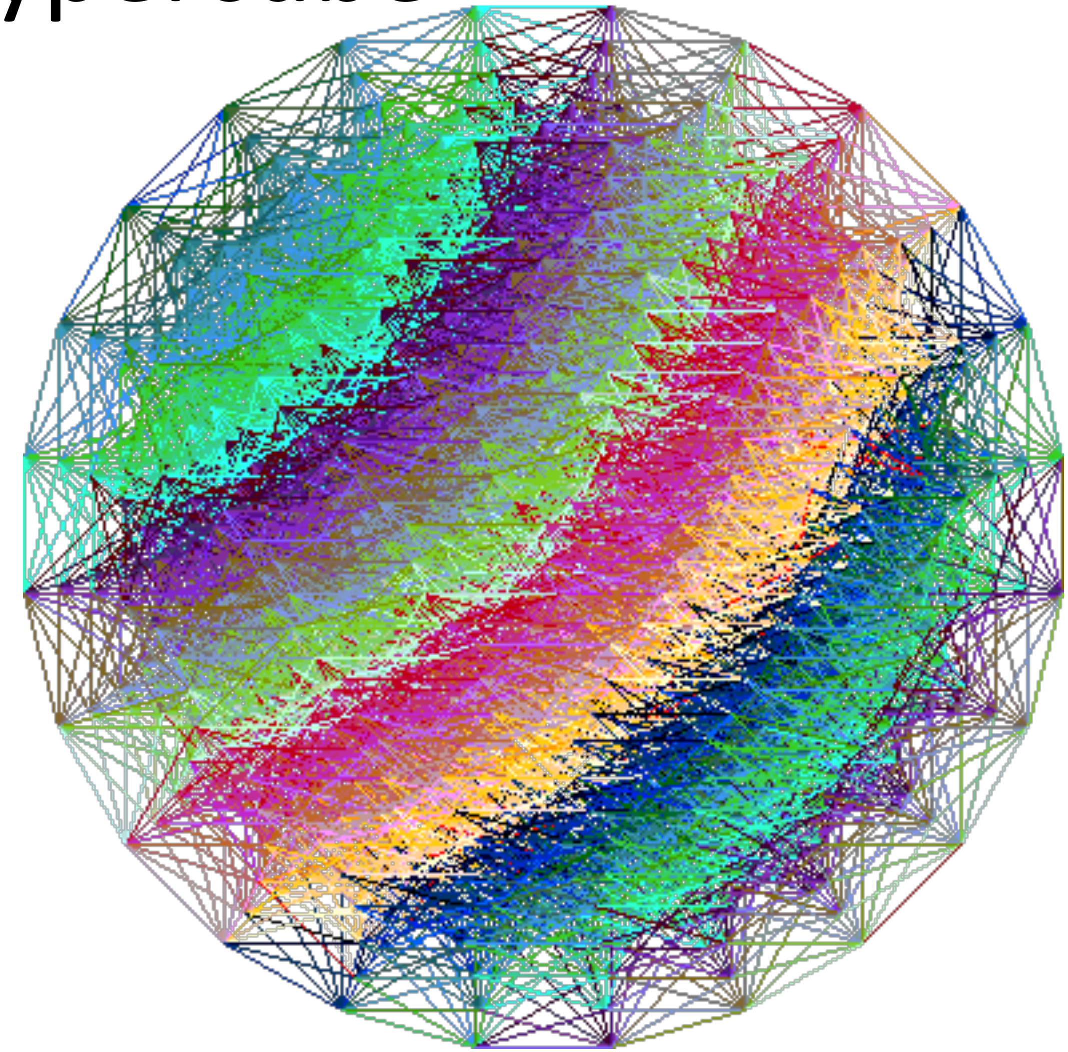
65k processors

can see CM-1 and CM-5 (from 1993) at Computer History Museum, Mountain View

http://www.inc.com/magazine/19950915/2622.html

Hypercube





PRAM

Abstract model of parallel computation

N processors accessing a single shared unbounded memory.

Often refined to Single Program Multiple Data (SPMD) or Single Instruction Multiple Data (SIMD)

See JaJa J.F. (2011) PRAM (Parallel Random Access Machines). In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-09766-4_23

NESL array operations

```
function factorial(n) =
  if (n <= 1) then 1
  else n*factorial(n-1);

{factorial(i) : i in [3, 1, 7]};
```

apply to each = parallel map (works with user-defined functions
=> load balancing)

list comprehension style notation

Online interpreter

The result of:

```
function factorial(n) =
    if (n <= 1) then 1
    else n*factorial(n-1);

{factorial(i) : i in [3, 1, 7]};
```

is:

```
factorial = fn : int -> int

it = [6, 1, 5040] : [int]

Bye.
```

Online interpreter

The result of:

```
function factorial(n) =

    if (n <= 1) then 1
    else n*factorial(n-1);


{factorial(i) : i in [3, 1, 7]};
```

is:

```
factorial = fn : int -> int

it = [6, 1, 5040] : [int]

Bye.
```

http://www.cs.cmu.edu/~scandal/nesl/alg-sequence.html

shows some interesting examples and gives access to the interpreter

http://www.cs.cmu.edu/~scandal/nesl/tutorial2.html

contains a tutorial, and the calls to the interpreter (via AWS) work for us

Parallel algorithms on sequences and strings

This page contains a collection of parallel algorithm on sequences and strings. It includes a brief desc

If you have arrived here via a search engine, we suggest going to the toplevel algorithms page.

Tree Scan

The algorithm we use is a standard tree-based algorithm that requires a total of *O(n)* work and *O(log*

```
{ a + b + c : a in [1,2,3,4,5]; b in [1,2,3,4,5]; c in [1,2,3,4]};

function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

Submit

Lots of examples to look at and play with!!

apply to each (multiple sequencs)

The result of:

```
{a + b : a in [3, -4, -9]; b in [1, 2, 3]};
```

is:

```
it = [4, -2, -6] : [int]

Bye.
```

apply to each (multiple sequencs)

The result of:

{a + b : a in [3, -4, -9]; b in [1, 2, 3]};

is:

it = [4, -2, -6] : [int]

Bye.

Qualifiers in comprehensions are zipping rather than nested as in Haskell
(Remember the parallel monad comprehensions that I showed you, however.)
Prelude> [a + b | a <- [3,-4,-9], b <- [1,2,3]]
[4,5,6,-3,-2,-1,-8,-7,-6]

Filtering too

The result of:

```
{a * a : a in [3, -4, -9, 5] | a > 0};
```

is:

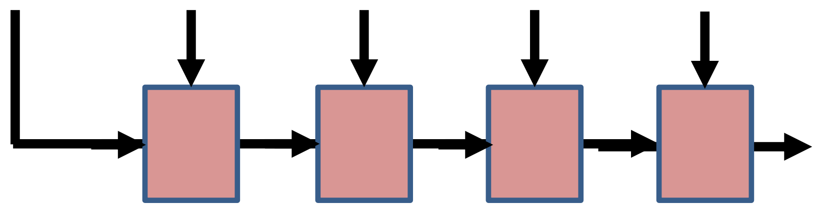
```
it = [9, 25] : [int]

Bye
```

scan (Haskell first)

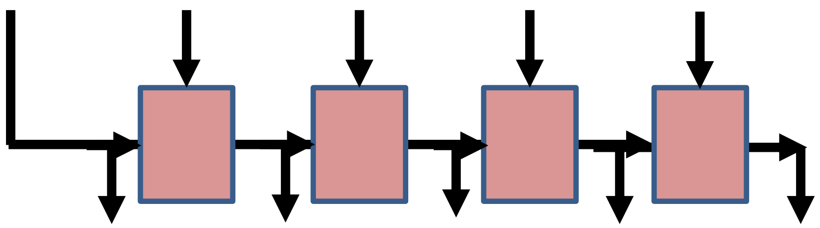
scanl1 :: (a -> a -> a) -> [a] -> [a]

```
*Main> scanl1 (+) [1..10]
[1,3,6,10,15,21,28,36,45,55]

*Main> scanl1 (*) [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

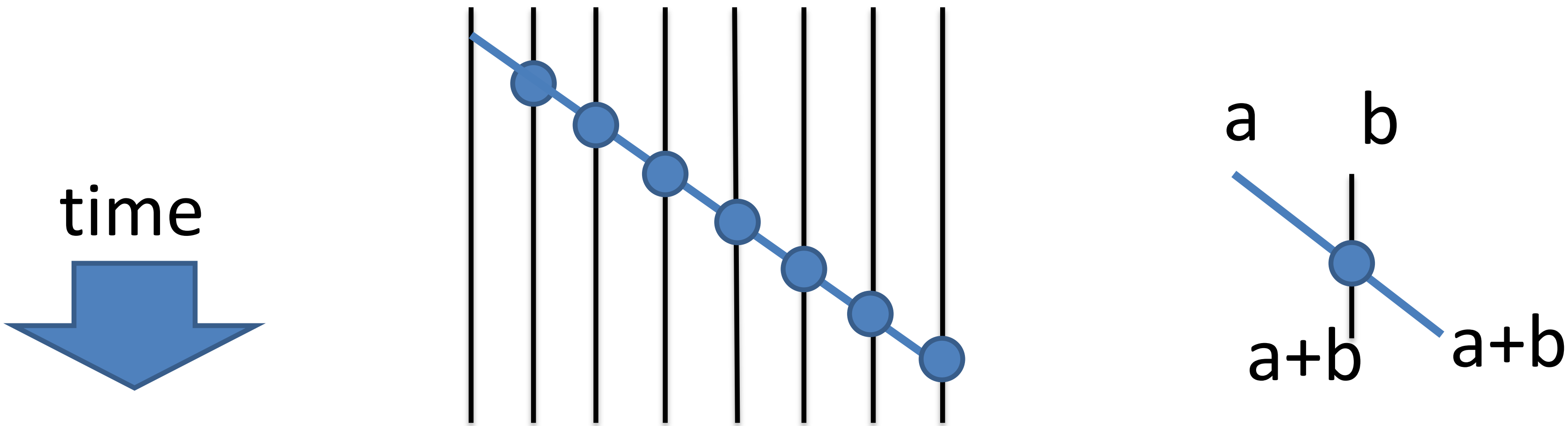
foldl1



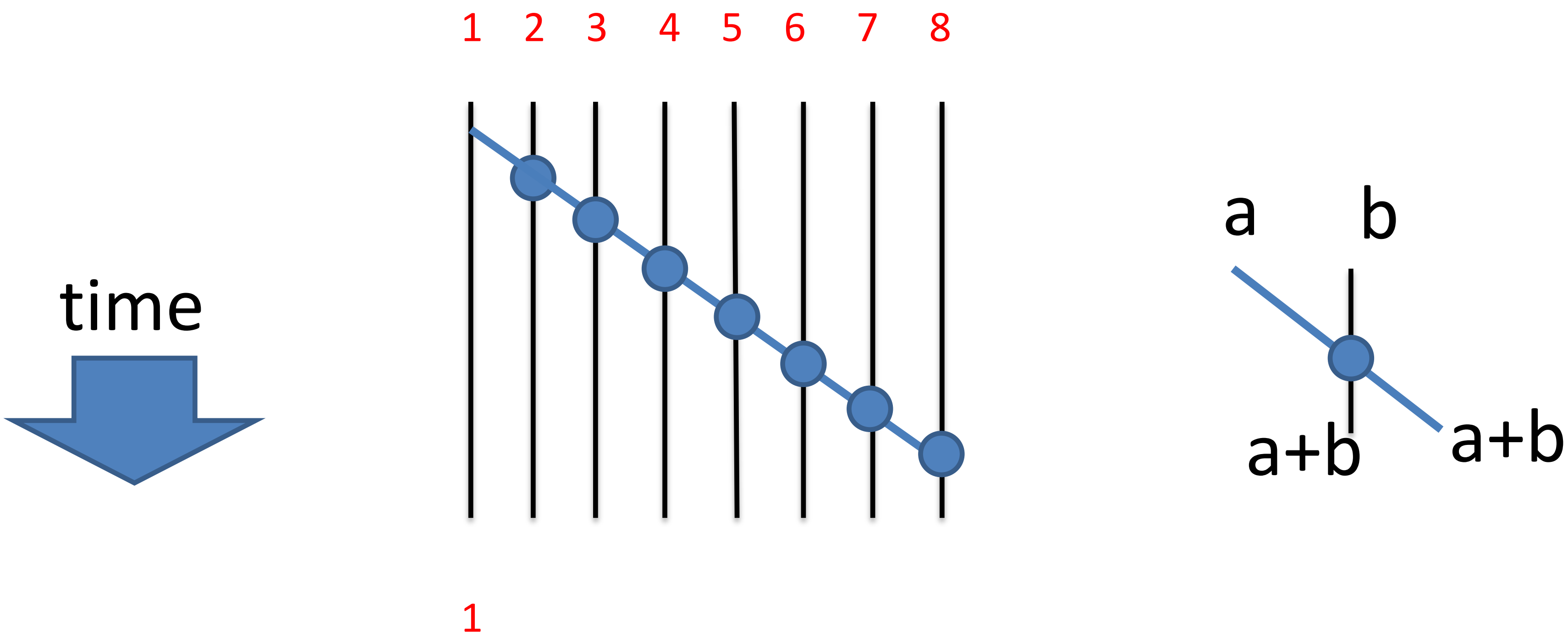
scanl1



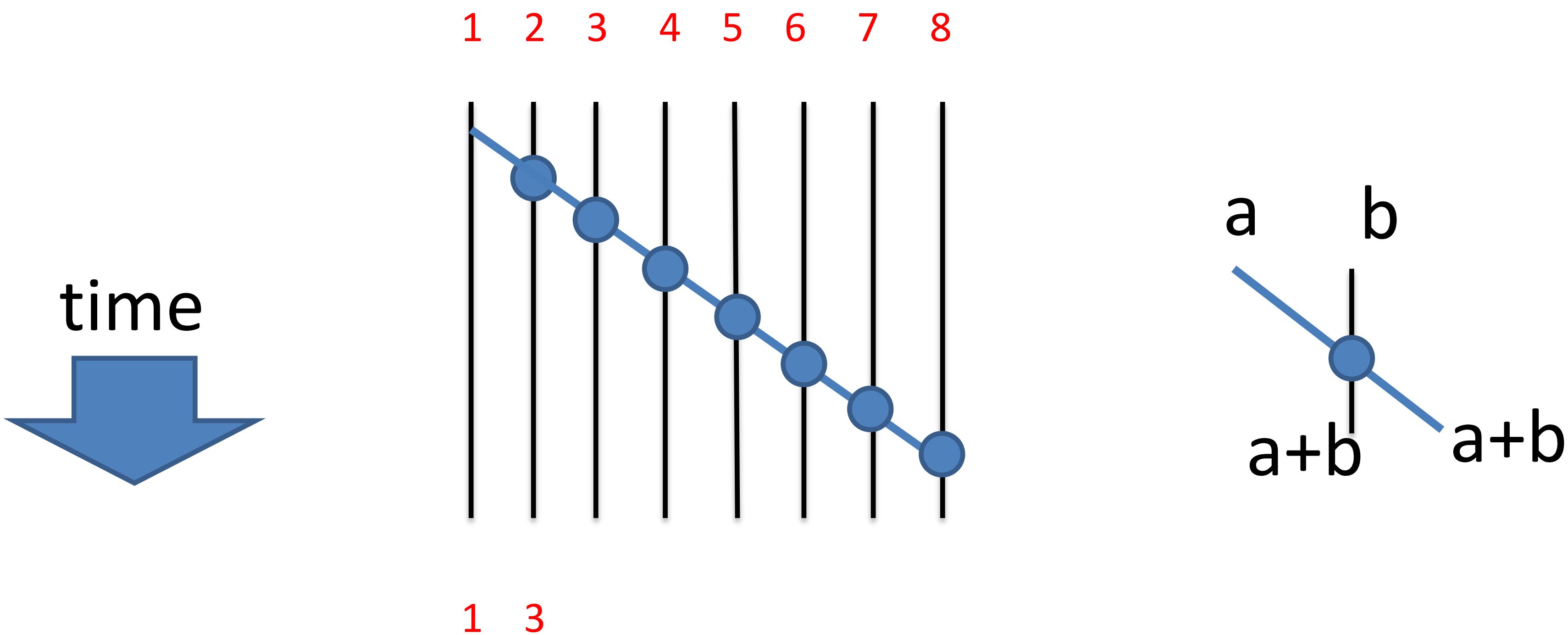
scan diagram (sequential)



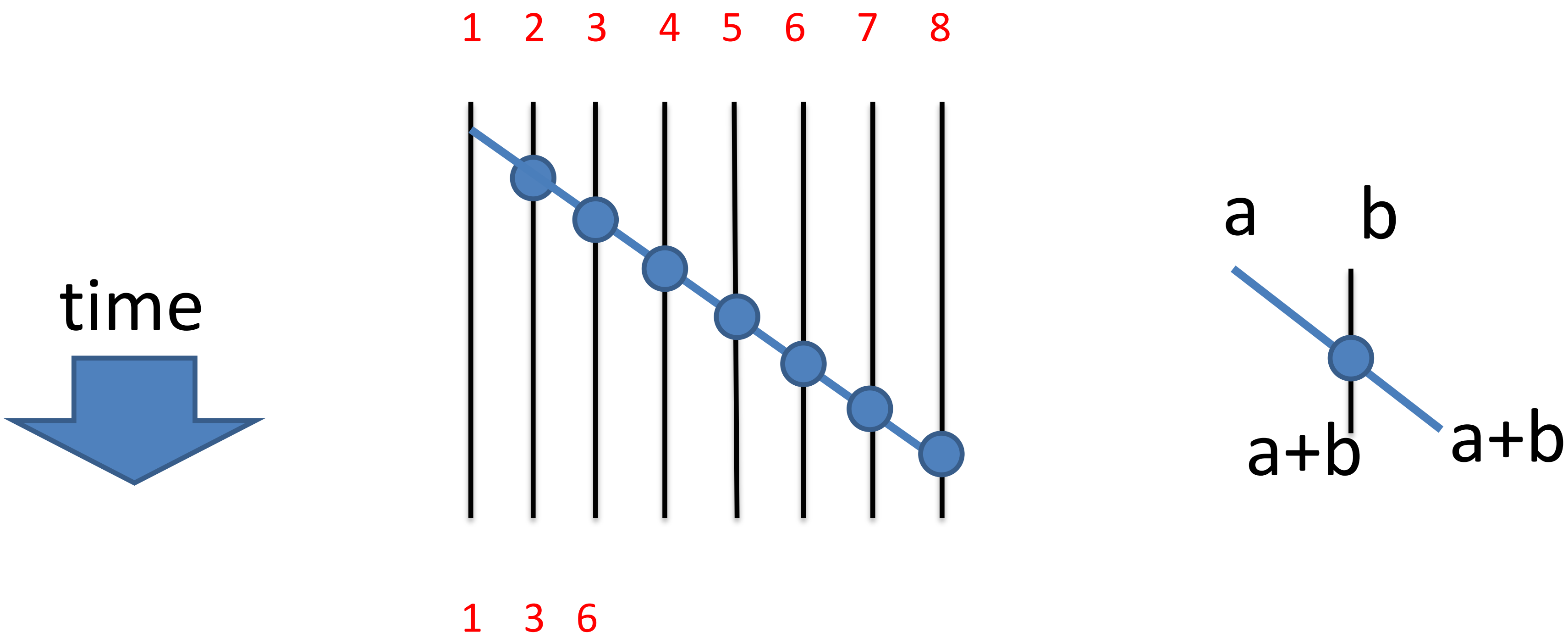
scan diagram (sequential)



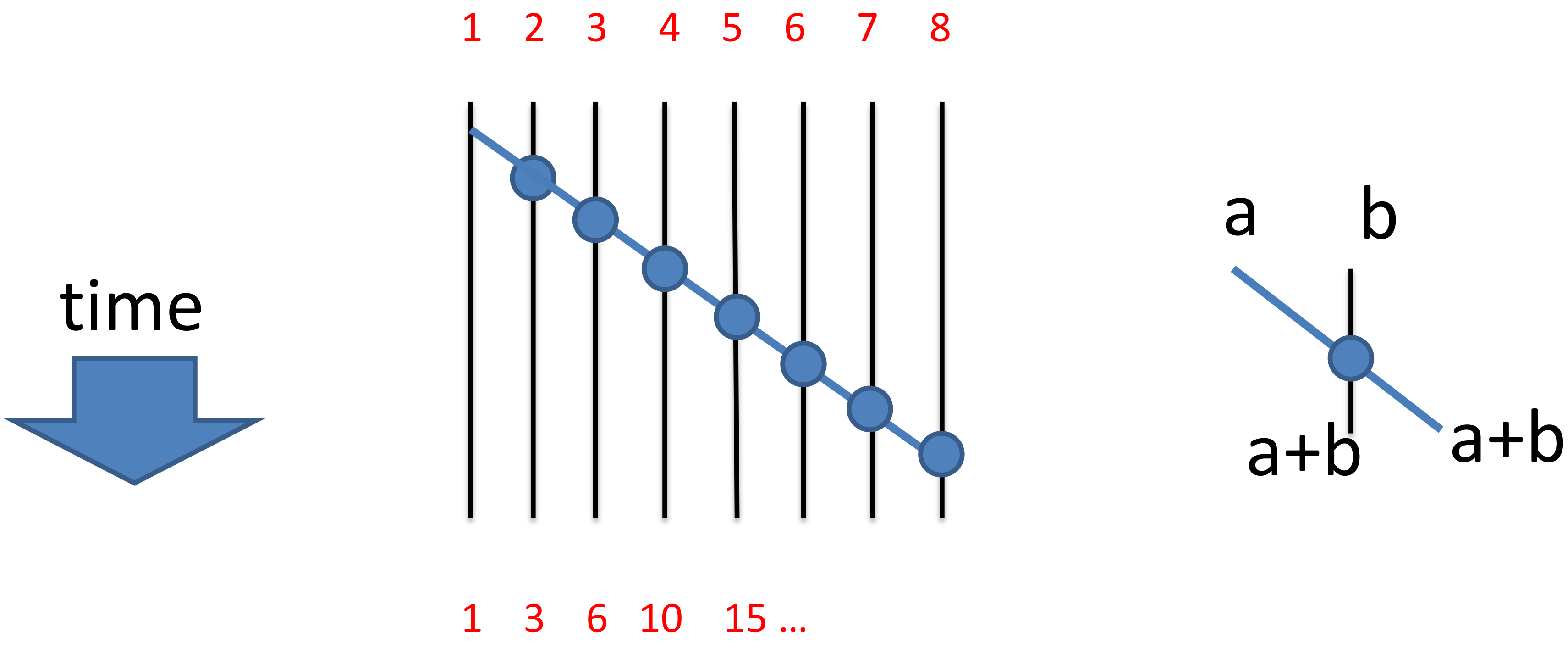
scan diagram (sequential)



scan diagram (sequential)

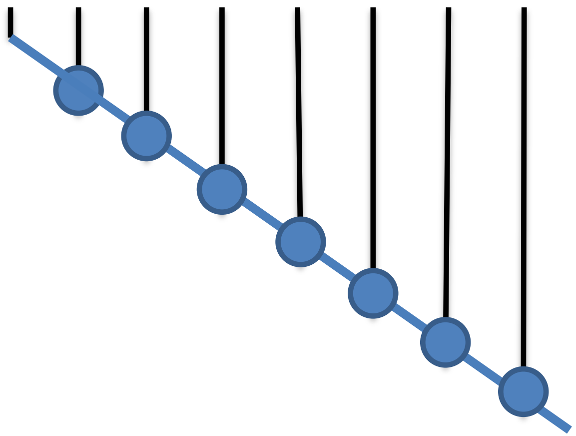


scan diagram (sequential)



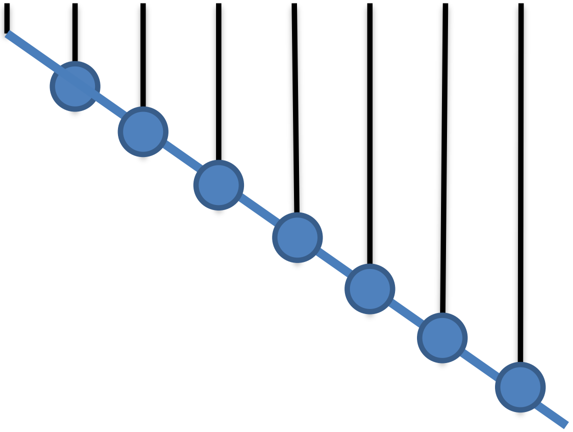
Associative operator enables parallelism

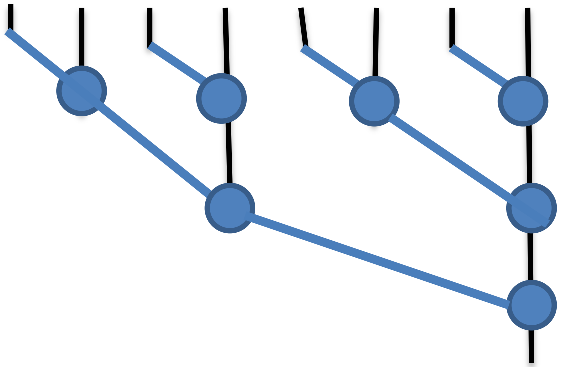
reduction



Associative operator enables parallelism

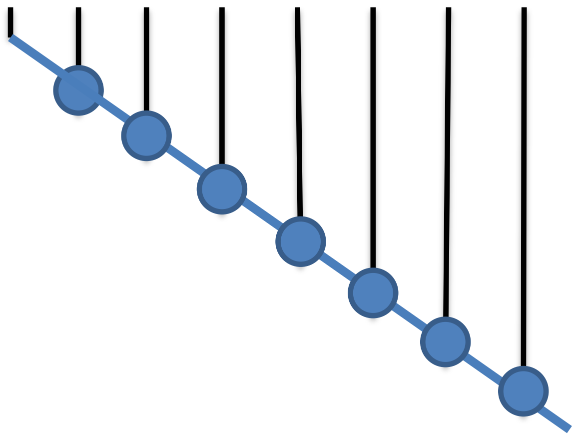
reduction



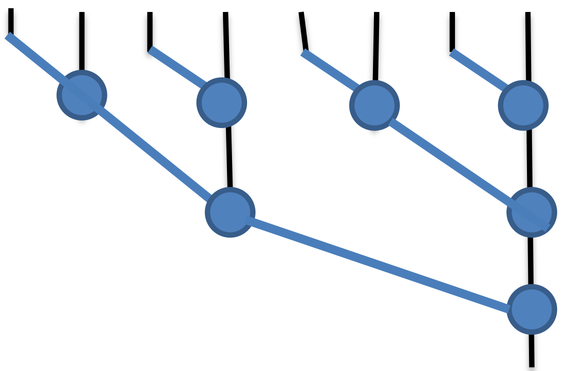


Associative operator enables parallelism

reduction

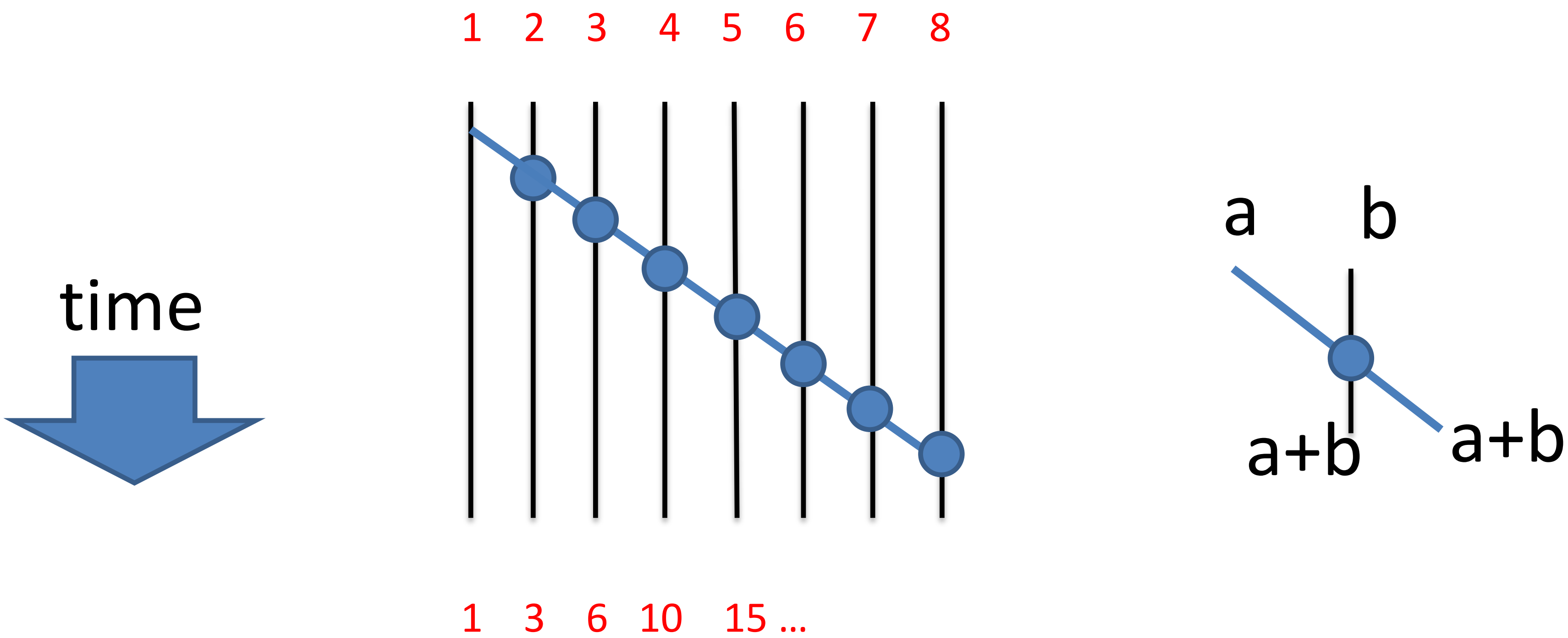


((((((a + b) + c) + d) + e) + f)+ g) + h

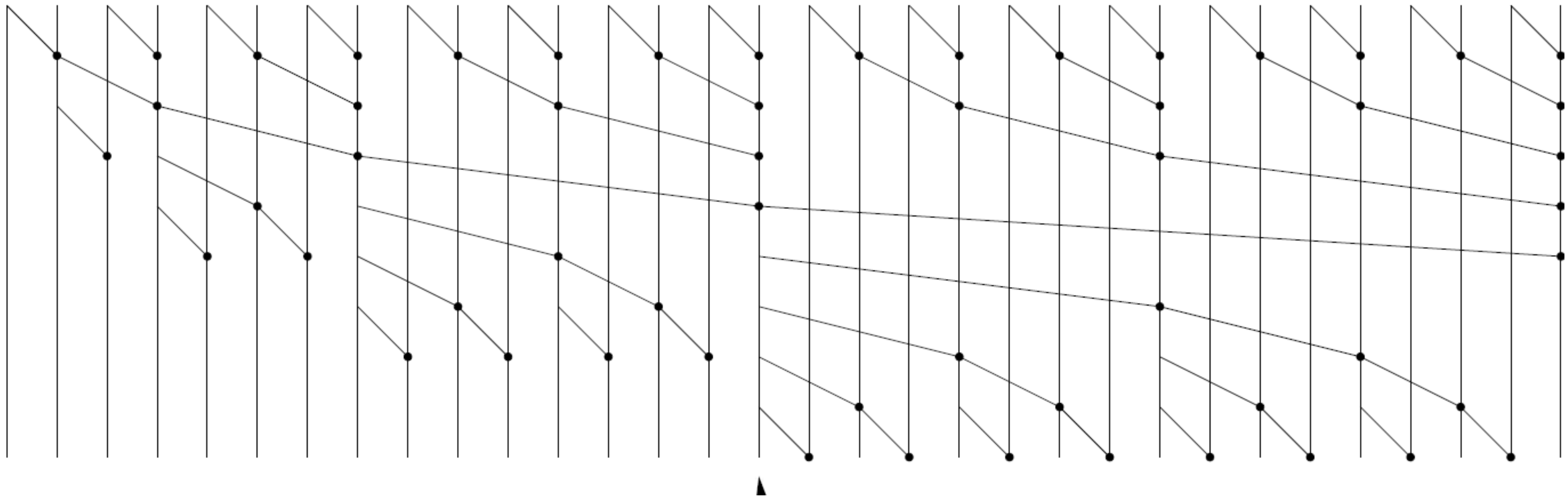


((a + b) + (c + d)) + ((e + f)+ (g + h))

scan diagram (sequential)



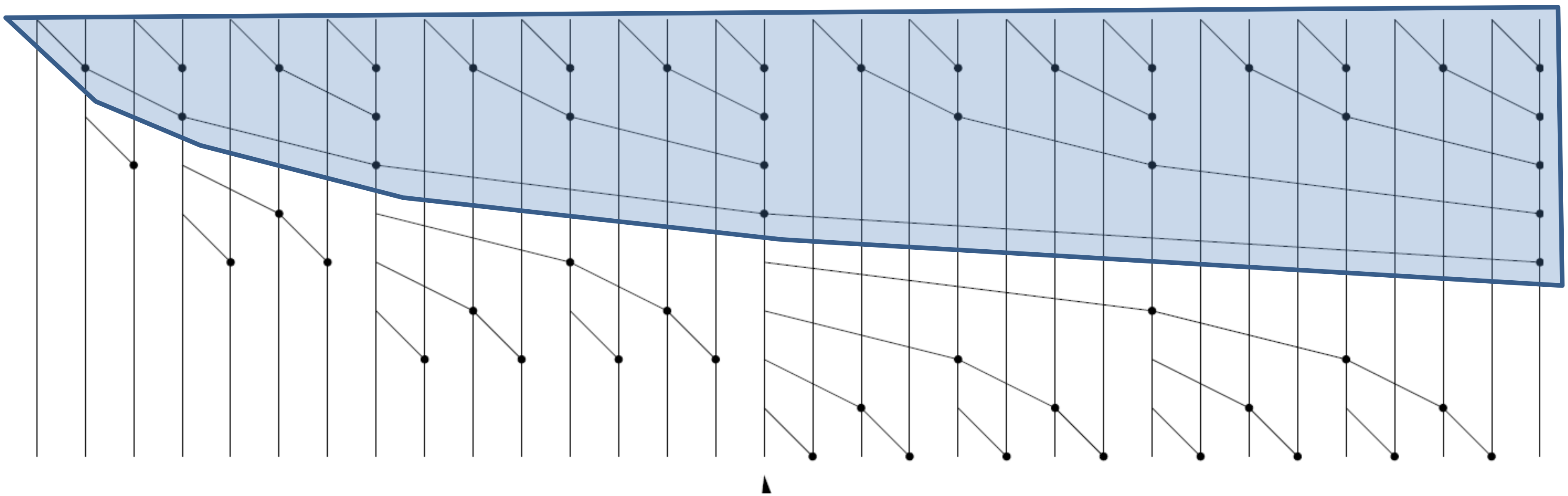
Brent Kung



1587

[A regular layout for parallel adders, IEEE Trans Comp 1982](#)

Brent Kung

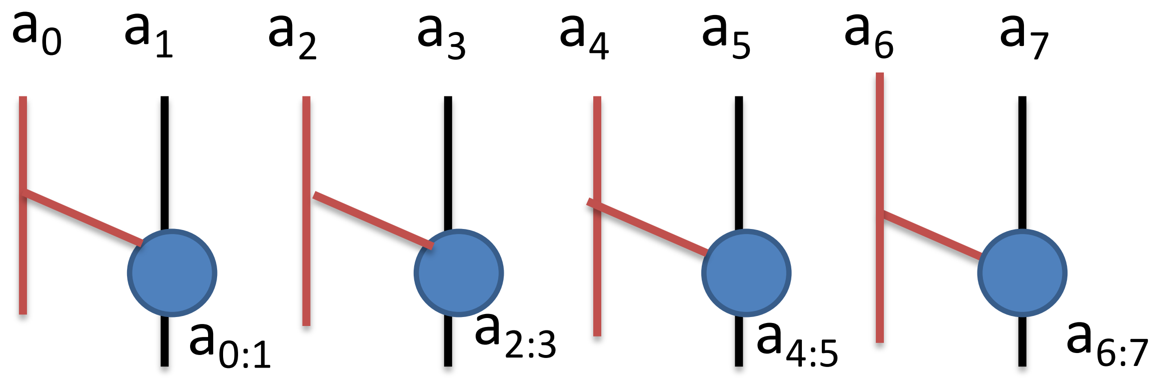


forward tree + several reverse trees

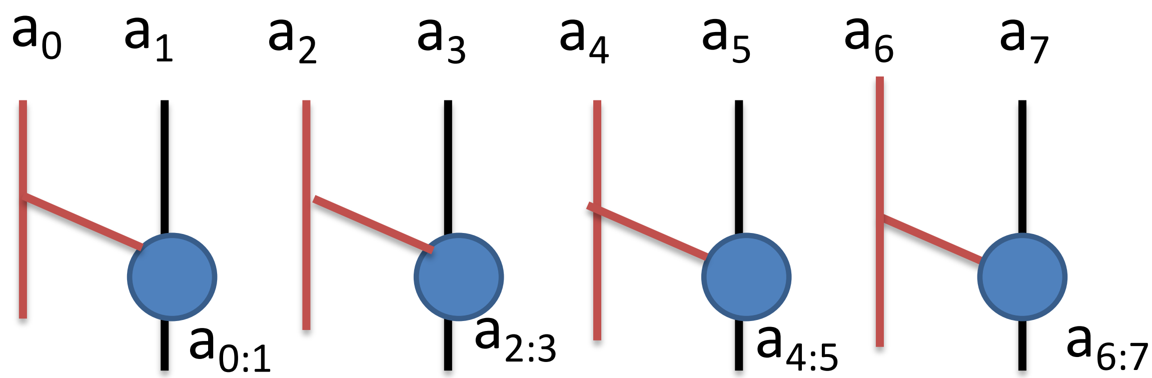
Puzzle (only for fun)

Can you figure out how to do scan (or parallel prefix) in depth exactly n for 2^n inputs ?

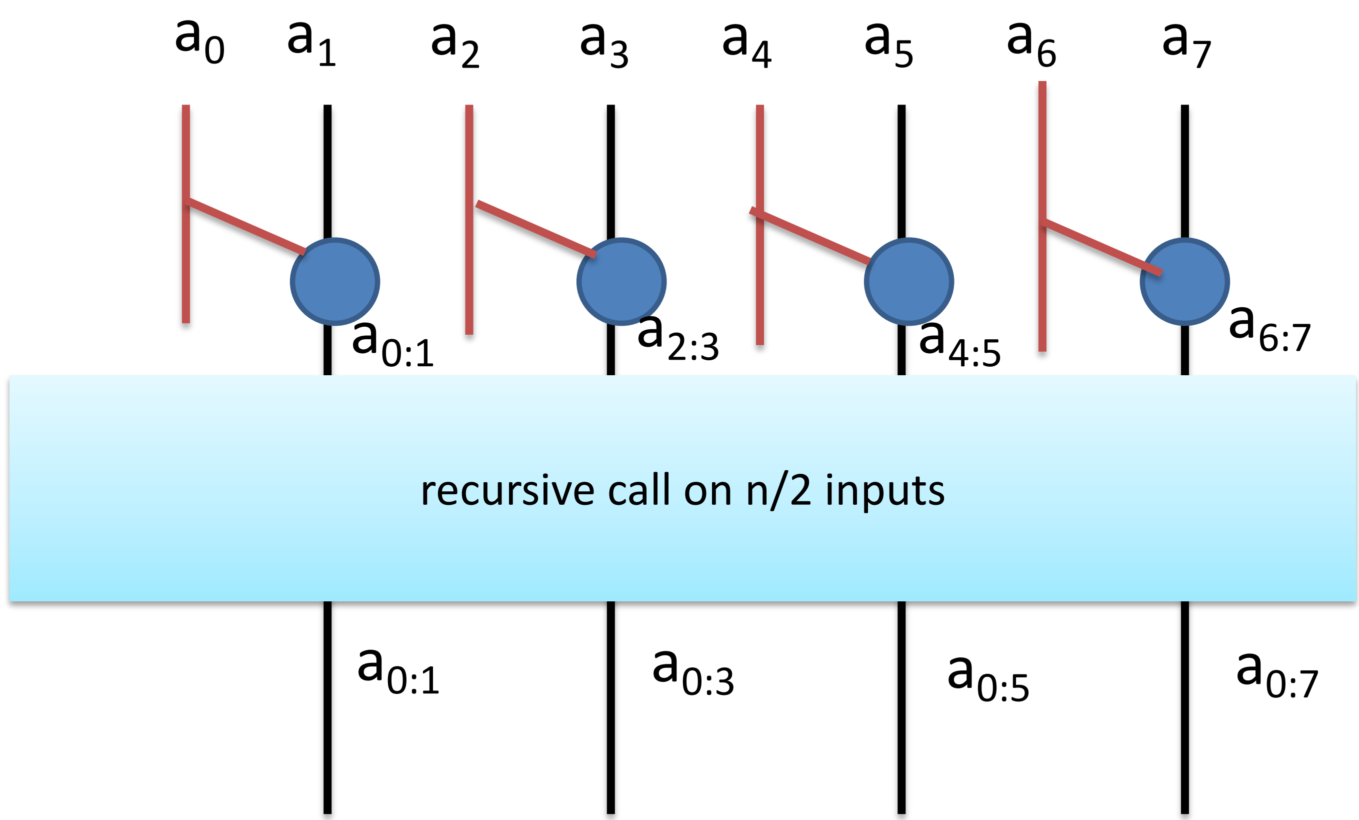
input sequence length 8



$a_{i:j}$ $\quad a_i + a_{i+1}\ldots + a_j$

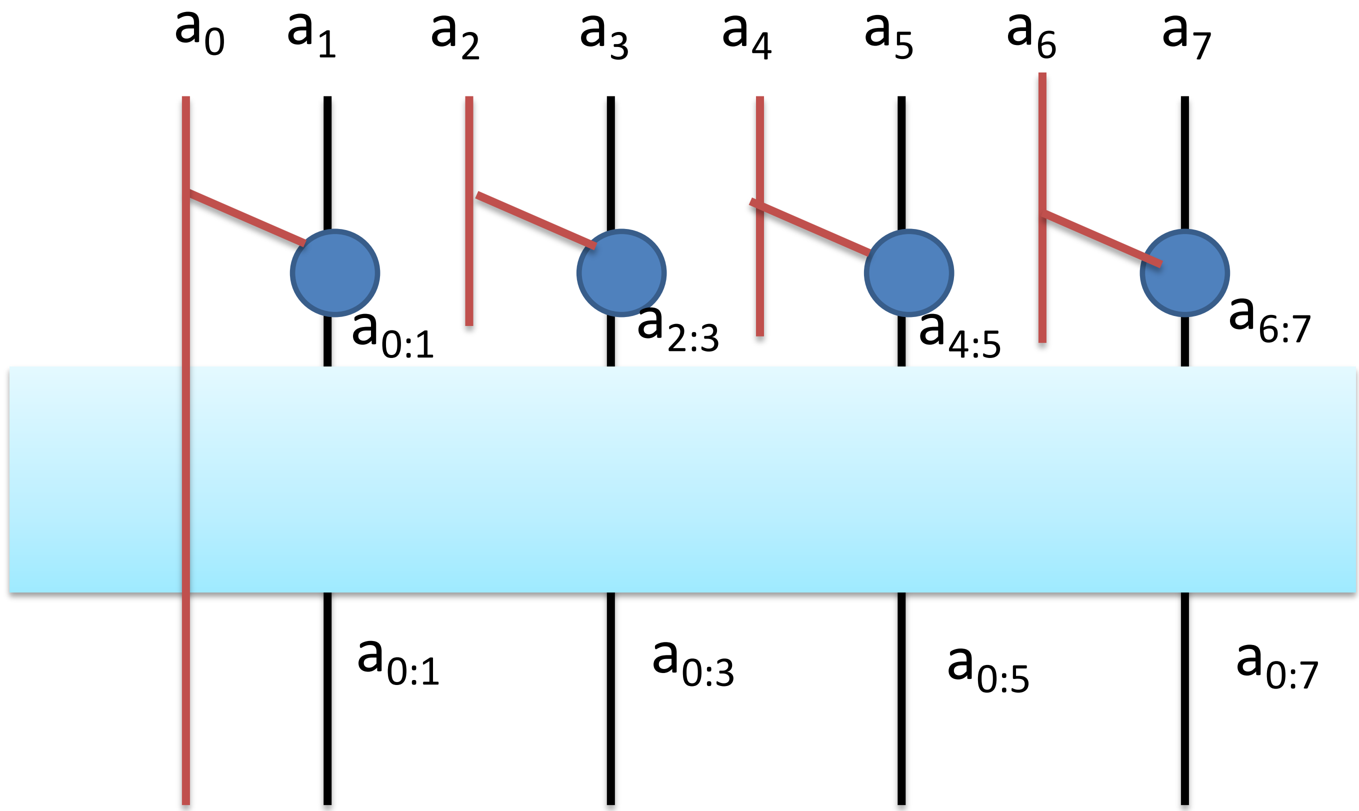


Brent Kung

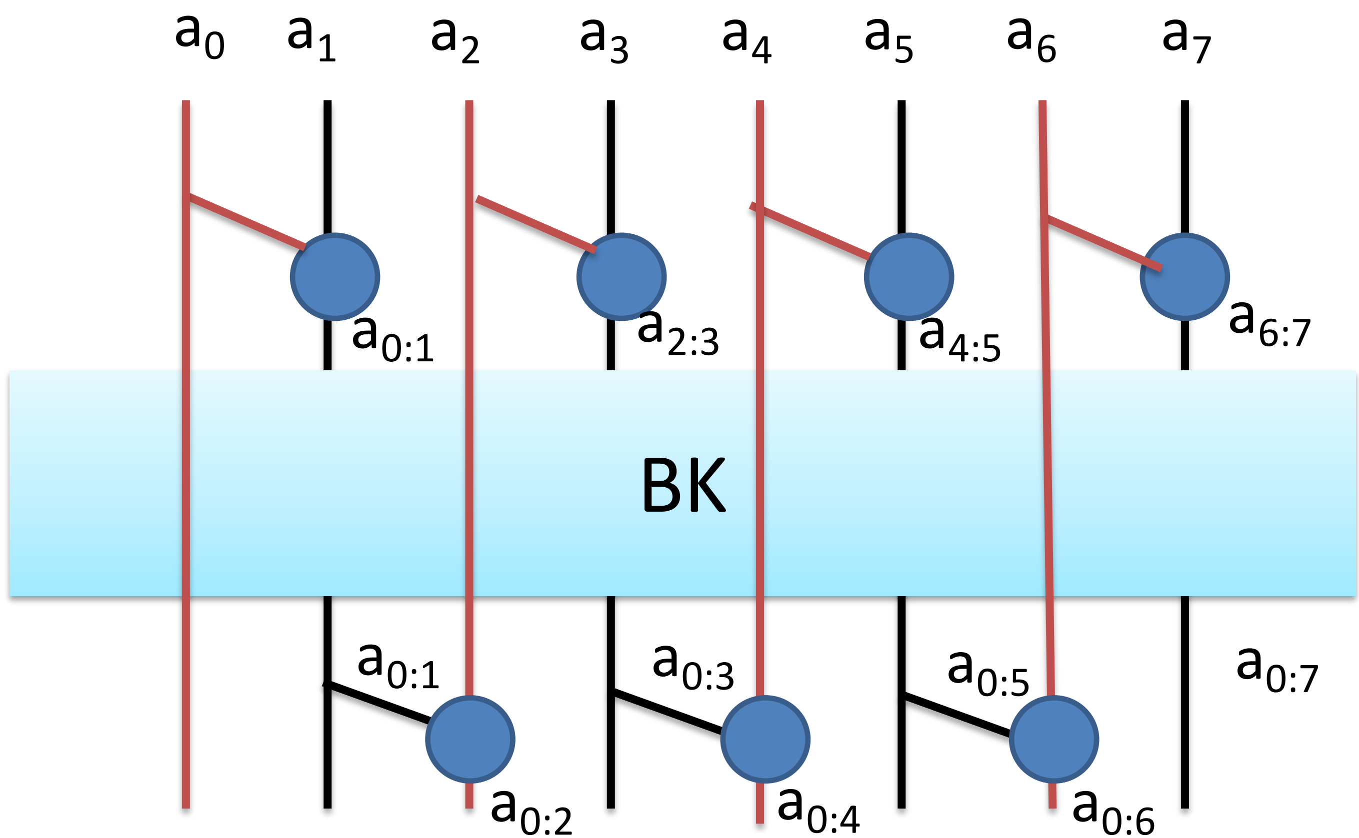


a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
$a_{0:1}$
$a_{2:3}$
$a_{4:5}$
$a_{6:7}$
recursive call on n/2 inputs
$a_{0:1}$
$a_{0:3}$
$a_{0:5}$
$a_{0:7}$

Brent Kung



Brent Kung



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
$a_{0:1}$
$a_{2:3}$
$a_{4:5}$
$a_{6:7}$
BK
$a_{0:1}$
$a_{0:3}$
$a_{0:5}$
$a_{0:7}$
$a_{0:2}$
$a_{0:4}$
$a_{0:6}$

Prescan (exclusive)

scan ”shifted right by one”

prescan of

$[a_1, a_2, a_3, a_4, \ . \ . \ . \ , a_n]$

is

$[i, \ a_1, \ a_1{*}a_2, \ a_1{*}a_2{*}a_3, \ \ldots, \ a_1{*} \ldots {*}a_{n-1}]$

identity element

scan from prescan

easy (constant time)

$[i, \quad a_1, \; a_1 * a_2, \; a_1 * a_2 * a_3, \ldots, a_1 * \ldots * a_{n-1}] \quad a_n$

$*$

$[a_1, \; a_1 * a_2, \; a_1 * a_2 * a_3, \ldots, a_1 * \ldots * a_{n-1}, a_1 * \ldots * a_n]$

scan from prescan

easy (constant ti e)

$[I, \quad a_1, \; a_1 * a_2, \; \ldots * a_2 * a_3, \ldots, a_1 * \ldots * a_{n-1}] \quad a_n$

$*$

$[a_1, \; a_1 * a_2, \; a_1 * \ldots a_3, \ldots, a_1 * \ldots * a_{n-1}, a_1 * \ldots * a_n]$

NOTE

parallel scan = parallel prefix

the power of scan

Blelloch pointed out that once you have scan
you can do LOTS of interesting algorithms, inc.

Lexically compare strings of characters. For example, to determine that "strategy" should appear before "stratification" in a dictionary

evaluate polynomials

solve recurrences e.g.

$$x_i = a_i x_{i-1} + b_i x_{i-2} \text{ and } x_i = a_i + b_i / x_{i-1}$$

implement radix sort

implement quicksort

solve tridiagonal linear systems

delete marked elements from an array

dynamically allocate processors

perform lexical analysis. For example, to parse a program into tokens

and many more

[Prefix sums and their applications]

CMU Tech report 1990

721

This chapter introduces one of the simplest and most useful building blocks for parallel algorithms: the all-prefix-sums operation.

prescan in NESL

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
       o = odd_elts(a);
       s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

prescan in NESL

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

zipWith op e o
zipWith op s e

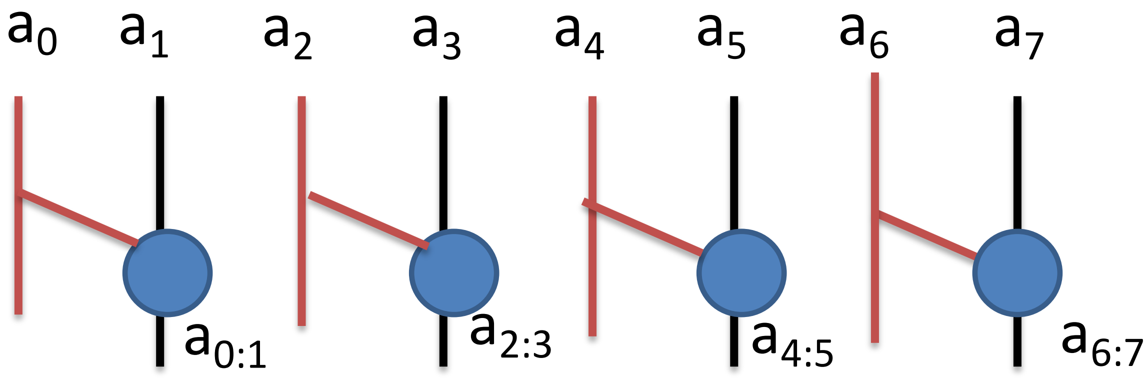
prescan

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

```
scan_op('+, 0, [2, 8, 3, -4, 1, 9, -2, 7]);

is:

scan_op = fn : ((b, b) -> b, b, [b]) -> [b] :: (a in any; b in any)

it = [0, 2, 10, 13, 9, 10, 19, 17] : [int]
```

http://www.cs.cmu.edu/~scandal/nesl/alg-sequence.html

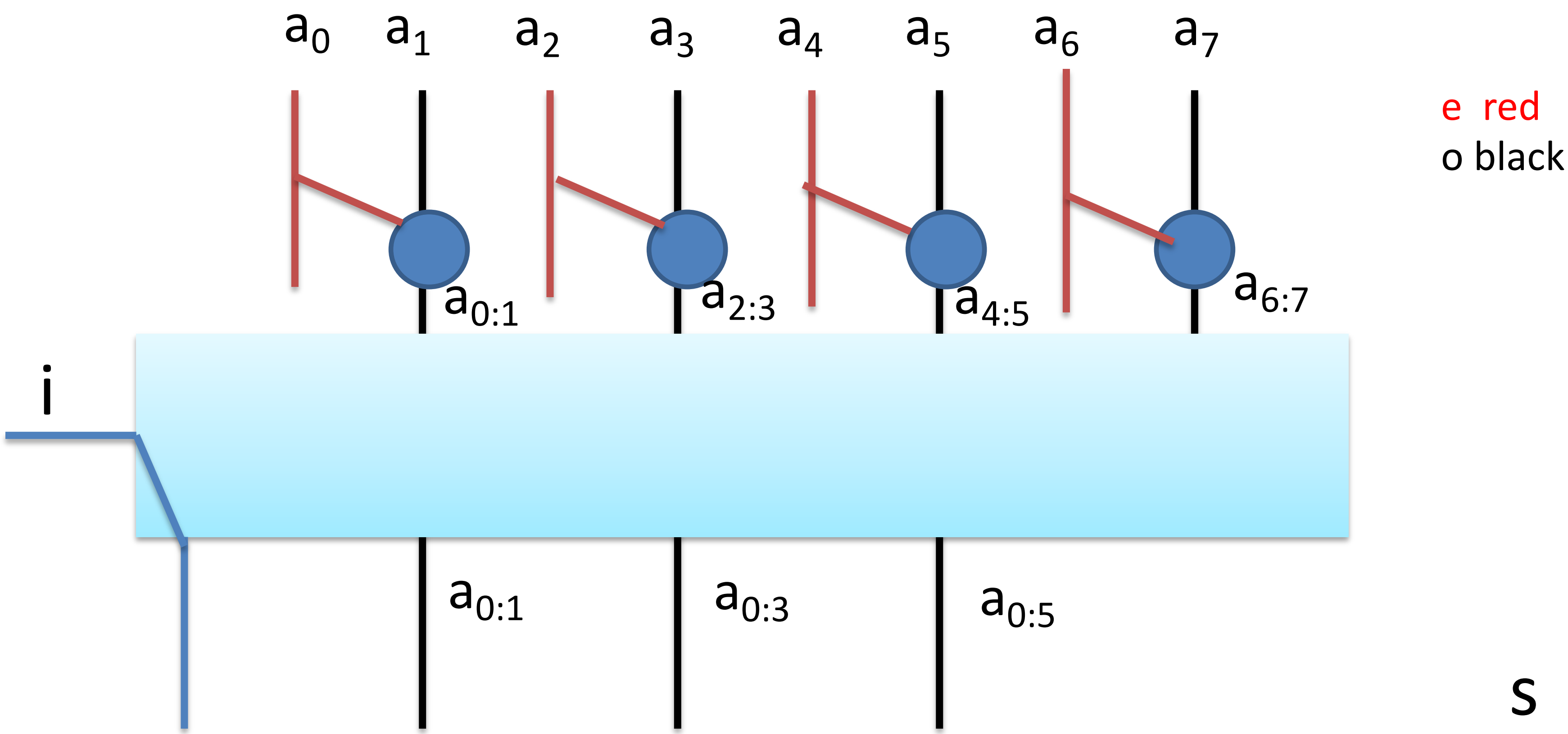
input sequence a, length 8



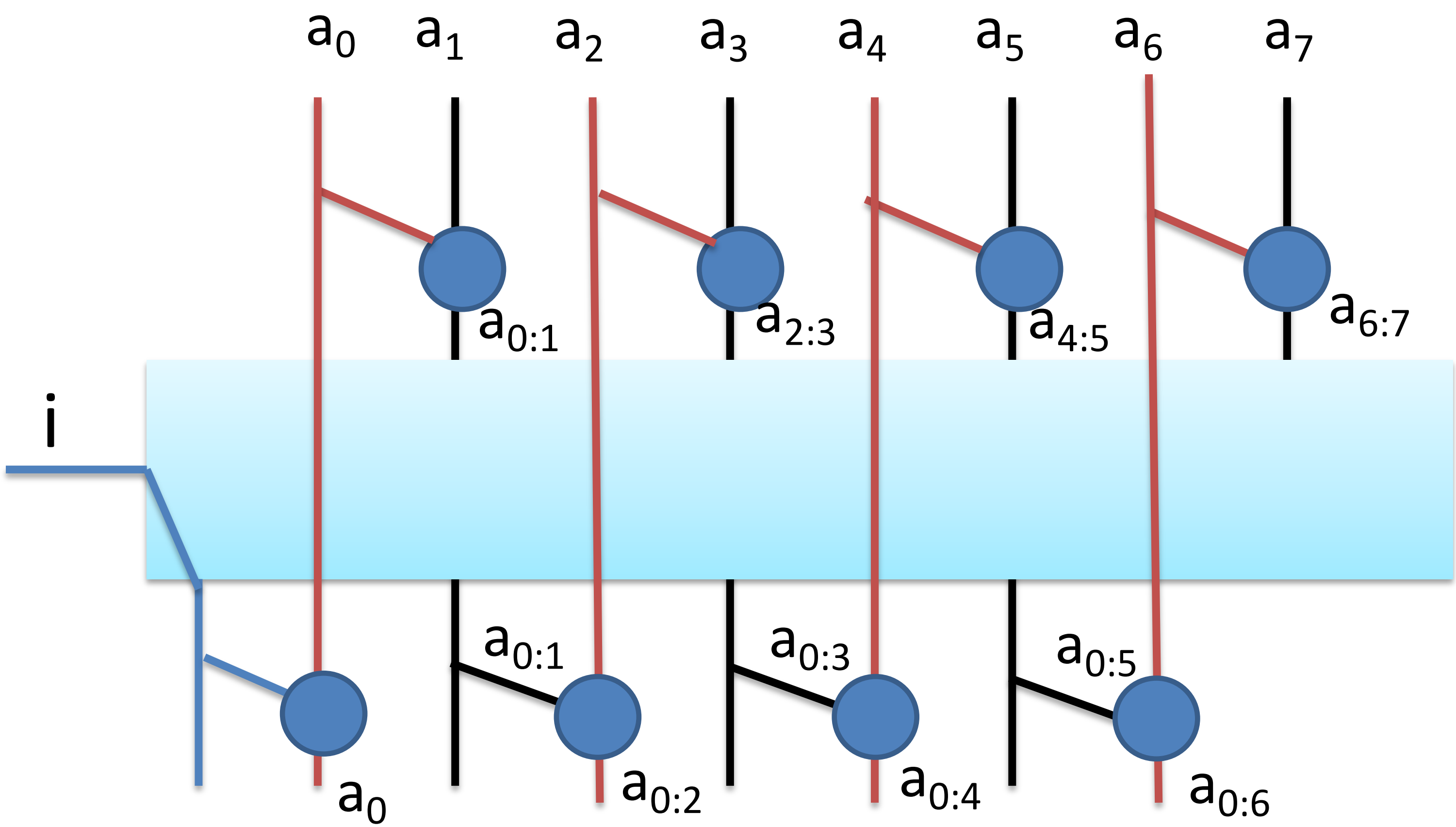
e red
o black

i

input sequence length 8



input sequence length 8



e red
o black

What does Nested mean??

```
{plus_scan(a) : a in [[2,3], [8,3,9], [7]]};
```

```
it = [[0, 2], [0, 8, 11], [0]] : [[int]]
```

What does Nested mean??

sequence of sequences
apply to each of a PARALLEL
function

```
{plus_scan(a) : a in [[2,3], [8,3,9], [7]]};
```

```
it = [[0, 2], [0, 8, 11], [0]] : [[int]]
```

What does Nested mean??

sequence of sequences
apply to each of a PARALLEL function

```
{plus_scan(a) : a in [[2,3], [8,3,9], [7]]};
```

```
it = [[0, 2], [0, 8, 11], [0]] : [[int]]
```

Implemented using Blelloch's Flattening Transformation, which converts nested parallelism into flat. Brilliant idea, challenging to make work in fancier languages (see DPH and good work on Manticore (ML))

What does Nested mean?? Another example

```
function svxv (sv, v) =
sum ({x * v[i] : (x, i) in sv});
```

```
function smxv (sm, v) =
{ svxv(row, v) : row in sm }
```

sparse vector sv, sequence of pairs of value and index
svxv   dot product of sparse vector and ordinary (dense) vector

sparse matrix sm, sequence of sparse vectors (rows)

this prescan is actually flat

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

Nestedness often from Divide and Conquer

```
function Quicksort(A) = if (#A < 2) then A else
                let pivot = A[#A/2];
                    lesser = {e in A| e < pivot};
                    equal = {e in A| e == pivot};
                    greater = {e in A| e > pivot};
                    result = {quicksort(v): v in [lesser,greater]};
                in result[0] ++ equal ++ result[1];
```

Nestedness is good for D&C and for irregular computations

What about a cost model?

Blelloch empasises

1) work : total number of operations

   represents total cost (integral of needed resources over time = running time on one processor)

2) depth or span: longest chain of sequential dependencies

   best possible running time on an unlimited number of processors

claims:

1) easier to think about algorithms based on work and depth than to use running time on machine with P processors (e.g. PRAM)
2) work and depth predict running time on various different machines

   (at least in the abstract)

work

on a sequential machine = sequential time

but can maybe be shared among multiple processors

Span

(or depth)

Allows analysis of extent to which work can be shared among processors

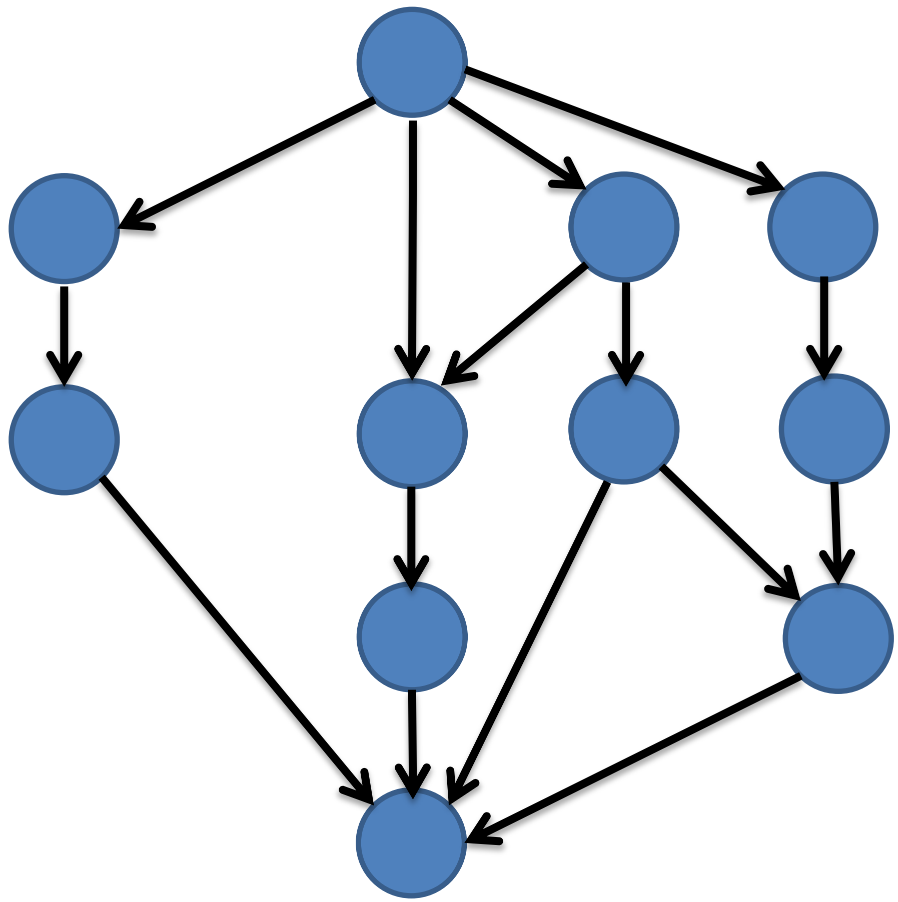
Span

(or depth)

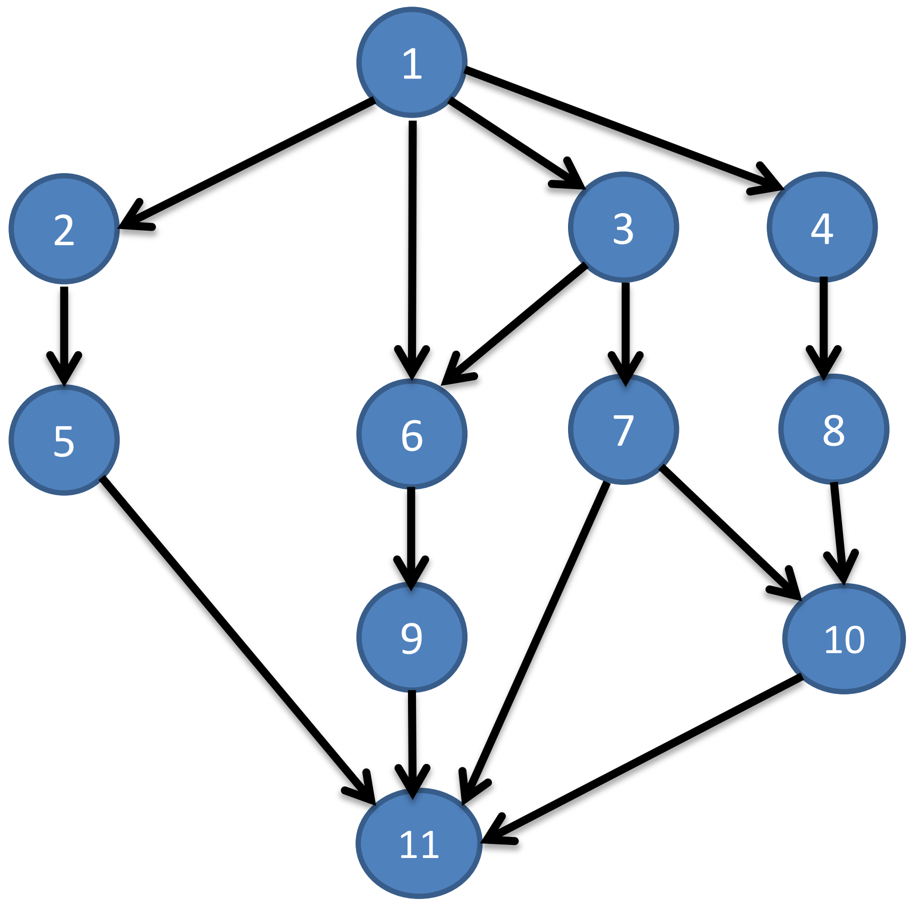
Allows analysis of extent to which work can be shared among processors

without resorting to details of machines, and how work is distributed over processors

Computation DAG

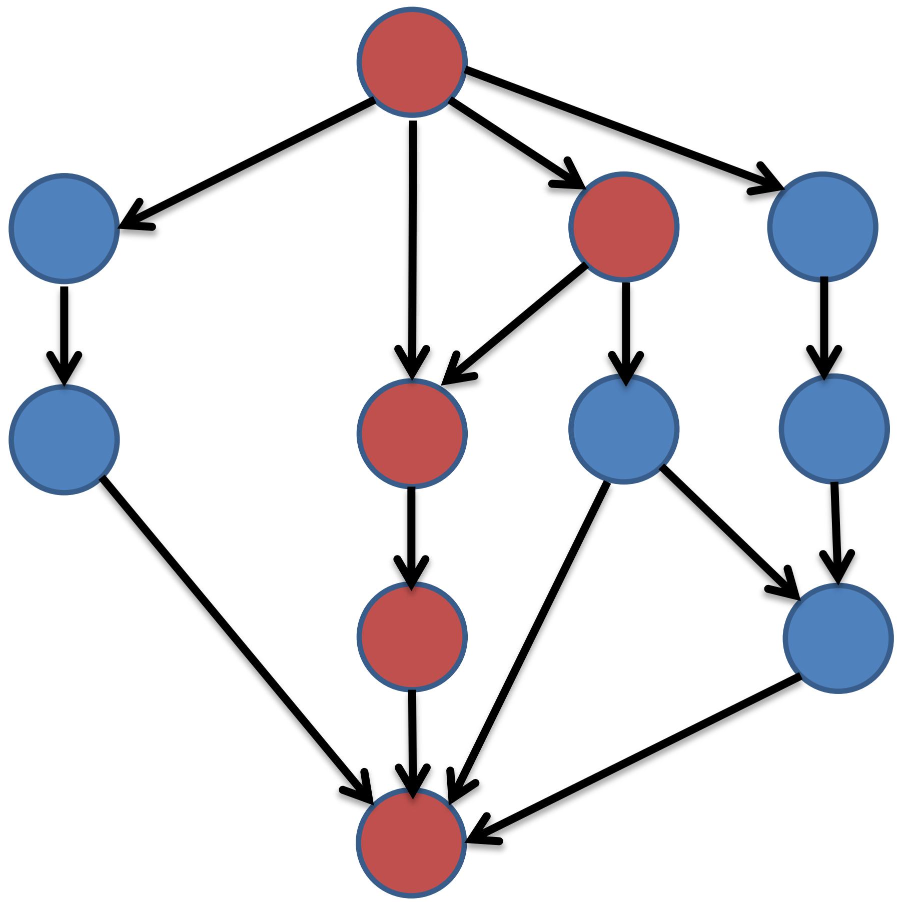


#ops, work, T_1



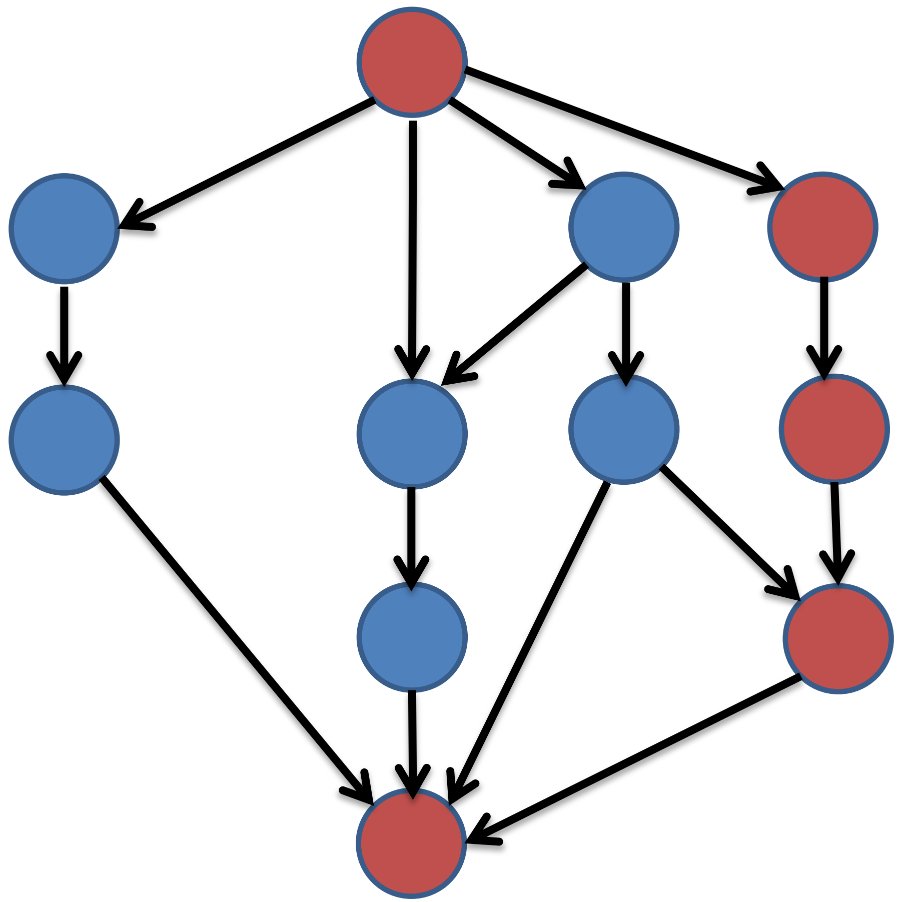
11

Depth, span, T_∞



5

Depth, span, T_∞



5

Lower bounds on T_p

Work

$T_p \geq T_1 / p$

Span

$T_p \geq T_\infty$

Lower bounds on T_p

Work

$T_p \geq T_1 / p$

Span

$T_p \geq T_\infty$

Work-Span

$T_p \geq \max(T_1 / p, T_\infty)$

What about an upper bound?

Brent's lemma (theorem)

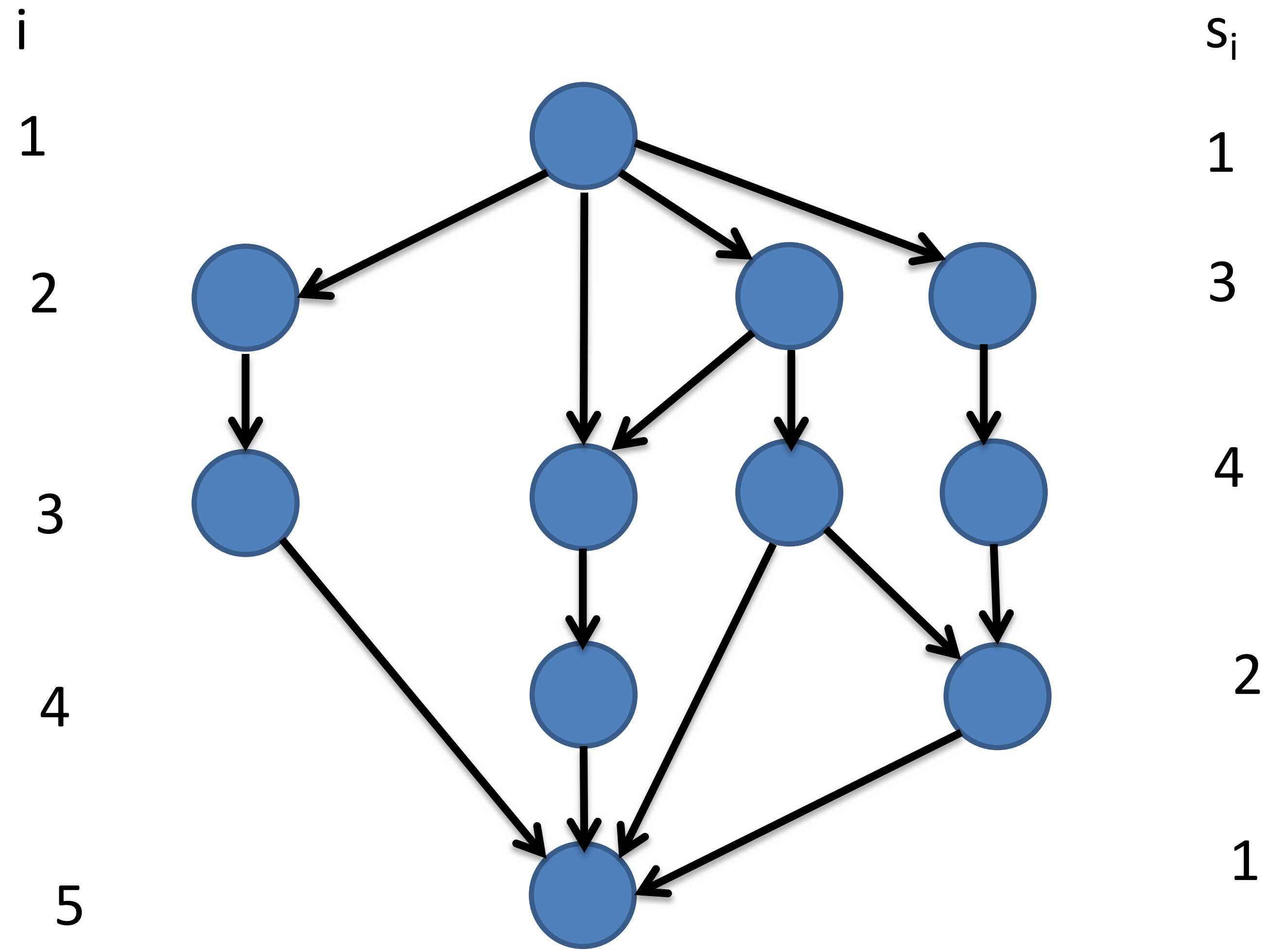
If a computation can be performed in t steps with q operations on a parallel computer (formally, a PRAM) with an unbounded number of processors, then the computation can be performed in $(q-t)/p + t$ steps with p processors

http://maths-people.anu.edu.au/~brent/pd/rpb022.pdf

JACM, 1974 5.5 pages! important part is 8 lines!

Why??

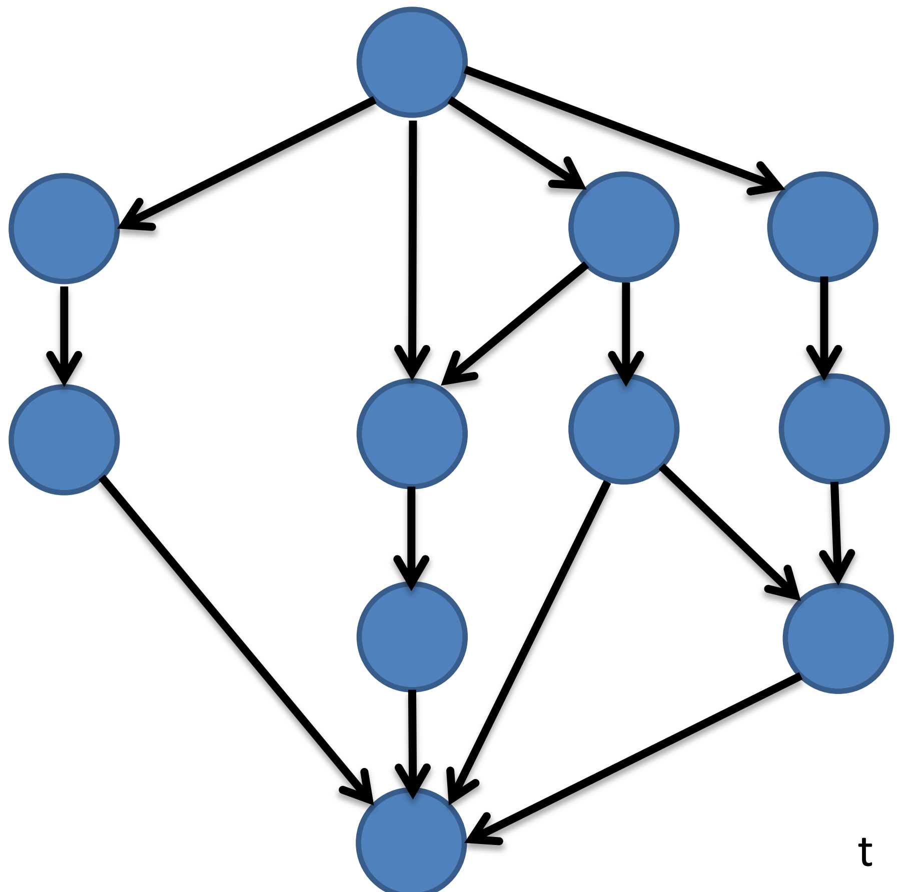
Suppose that s_i operations are performed at time i for i = 1, 2, .. t with an unbounded number of processors


i
1
2
3
4
5
s_i
1
3
4
2
1

Suppose that s_i operations are performed at time i for i = 1, 2, .. t with an unbounded number of processors

Then q = time on one processor = total number of operations

$$= \sum_{i=1}^{t} s_i$$



i	s_i
1	1
2	3
3	4
4	2
5	1

$$\sum_{i=1}^{t} s_i \quad = \quad \text{total work}$$

Using p processors, we can simulate time step i in time

$$\left\lceil s_i / p \right\rceil \leq \frac{s_i - 1}{p} + 1$$

Using p processors, we can simulate time step i in time

$$\left\lceil s_i / p \right\rceil \leq \frac{s_i - 1}{p} + 1$$

because ceiling (n/m) = floor((n-1)/m) +1 for m positive

Total time on p processors

$$\leq \sum_{i=1}^{t} \lceil s_i / p \rceil$$

$$\leq \sum_{i=1}^{t} \left(\frac{s_i - 1}{p} + 1 \right)$$

$$= (q-t)/p + t$$

Brent’s theorem (again)

On p processors, a parallel computation can be performed in time T_p where

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty$$

scheduler

A greedy scheduler guarantees that no processor will be idle (= not working on part of the computation) if there is work remaining to do

A good scheduler gives behavior according to Brent’s theorem

Parallelism

T_1 / T_∞

Average amount of work needing to be done at each step along the span

Gives a bound on the possible speedup

Back to NESL (cost model)

$$W(e1 + e2) = 1 + W(e1) + W(e2)$$

Back to NESL (cost model)

$$W(e1 + e2) = 1 + W(e1) + W(e2)$$

$$D(e1 + e2) = 1 + \max(D(e1), D(e2))$$

Back to NESL (cost model)

$W(e1 + e2) = 1 + W(e1) + W(e2)$

$D(e1 + e2) = 1 + \max(D(e1), D(e2))$

Work adds

Depth involves max when 2 expressions can be evaluated in parallel

$$W(\{e1(a) : a \text{ in } e2\}) = 1 + W(e2) + \sum_{a \text{ in } e2} W(e1(a))$$

$$W(\{e1(a) : a \text{ in } e2\}) = 1 + W(e2) + \sum_{a \text{ in } e2} W(e1(a))$$

$$D(\{e1(a) : a \text{ in } e2\}) = 1 + D(e2) + \max_{a \text{ in } e2} D(e1(a))$$

From the NESL quick reference

Basic Sequence Functions

Basic Operations	Description	Work	Depth
#a	Length of a	O(1)	O(1)
a[i]	ith element of a	O(1)	O(1)
dist(a,n)	Create sequence of length n with a in each element.	O(n)	O(1)
zip(a,b)	Elementwise zip two sequences together into a sequence of pairs.	O(n)	O(1)
[s:e]	Create sequence of integers from s to e (not inclusive of e)	O(e-s)	O(1)
[s:e:d]	Same as [s:e] but with a stride d.	O((e-s)/d)	O(1)

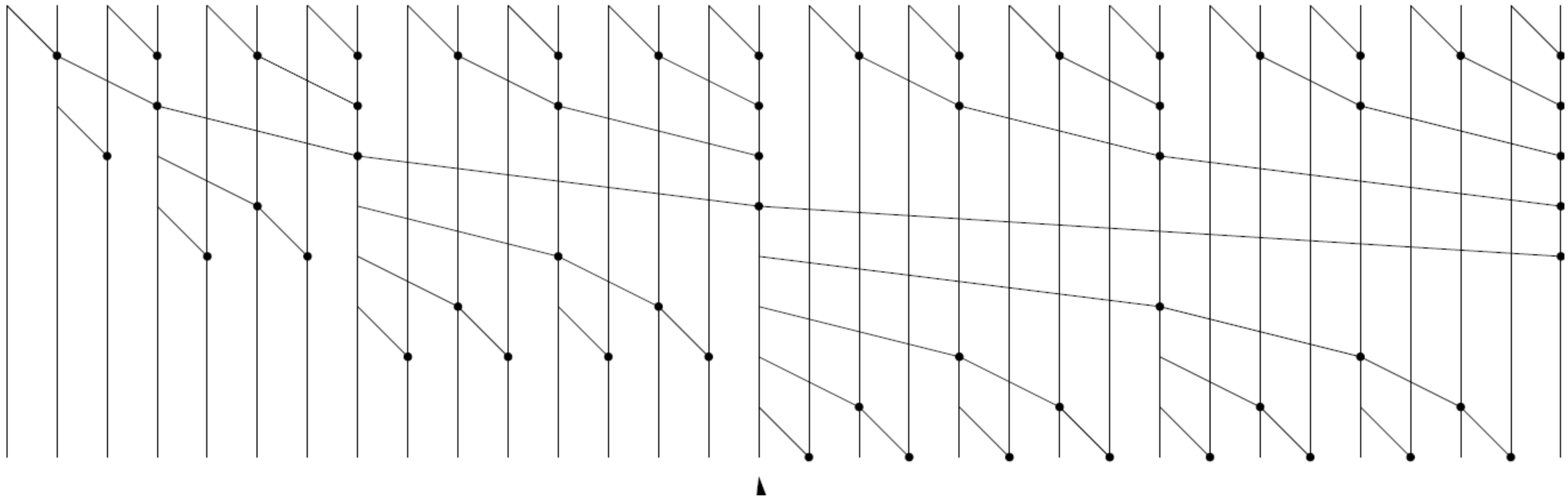
Scans

plus_scan(a)	Execute a scan on a using the + operator	O(n)	O(log n)
min_scan(a)	Execute a scan on a using the minimum operator	O(n)	O(log n)
max_scan(a)	Execute a scan on a using the maximum operator	O(n)	O(log n)
or_scan(a)	Execute a scan on a using the or operator	O(n)	O(log n)
and_scan(a)	Execute a scan on a using the and operator	O(n)	O(log n)

NESL

For modeling the cost of NESL we augment a standard call by value operational semantics to return two cost measures: a DAG representing the sequential dependences in the computation and a measure of the space taken by a sequential implementation. We show that a NESL program with w work (nodes in the DAG) d depth (levels in the DAG) and s sequential space can be implemented on a p processor butterfly network, hypercube or CRCW PRAM using O(w/p + d log p) time and O (s + dp log p) reachable space. For programs with sufficient parallelism these bounds are optimal in that they give linear speedup and use space within a constant factor of the sequential space.

Back to our scan



oblivious or data independent computation

$N = 2^k$ inputs, work of dot is 1

depth = ?
work = ?

$$\text{depth} = 2k - 1$$

$$N = 2^k$$

$$\text{work} = \sum_{i=0}^{k} (2^i - 1) = (2^{k+1} - 1) - (k+1)$$

$$= 2N - k - 2$$

Depth is O(k) and work is O(N)

Question

What does this cost model NOT cover?

NESL : what more should be done?

Take account of LOCALITY of data and account for communication costs (Blelloch has been working on this.)

Deal with exceptions and randomness

Find ways to back out of parallelism (flattening too aggressive)

(See retrospective slides from Blelloch, 2006)

NESL also influenced

Futhark, which you will see on thursday and use in the lab (T. Henriksen)

The Java 8 streams May 4 (P. Sestoft)

Data Parallel Haskell and Accelerate May 5 (G. Keller)

Single Assignment C
Intel Array Building Blocks (ArBB)
That has been retired, but ideas are reappearing as C/C++ extensions
and many others

Collections seem to encourage a functional style even in non functional languages
(remember Backus' paper from first lecture)

Summary

Programming-based cost models are (according to Blelloch) MUCH BETTER
than machine-based models

They open the door to other kinds of abstract costs than just work, depth, space …

There is fun to be had with parallel functional algorithms (especially as the
Algorithms community is still struggling to agree on useful models for use
In analysing parallel algorithms).

Read Blelloch's papers! They are great.

End