Parallel Functional Programming
Data Parallelism

Mary Sheeran

http://www.cse.chalmers.se/edu/course/pfp

Data parallelism

Introduce parallel data structures and make operations on them parallel

Often data parallel arrays

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

Data parallelism

Introduce parallel data structures and make operations on them parallel

Often data parallel arrays

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

See video of ICFP10 invited talk on Lectures page

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

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

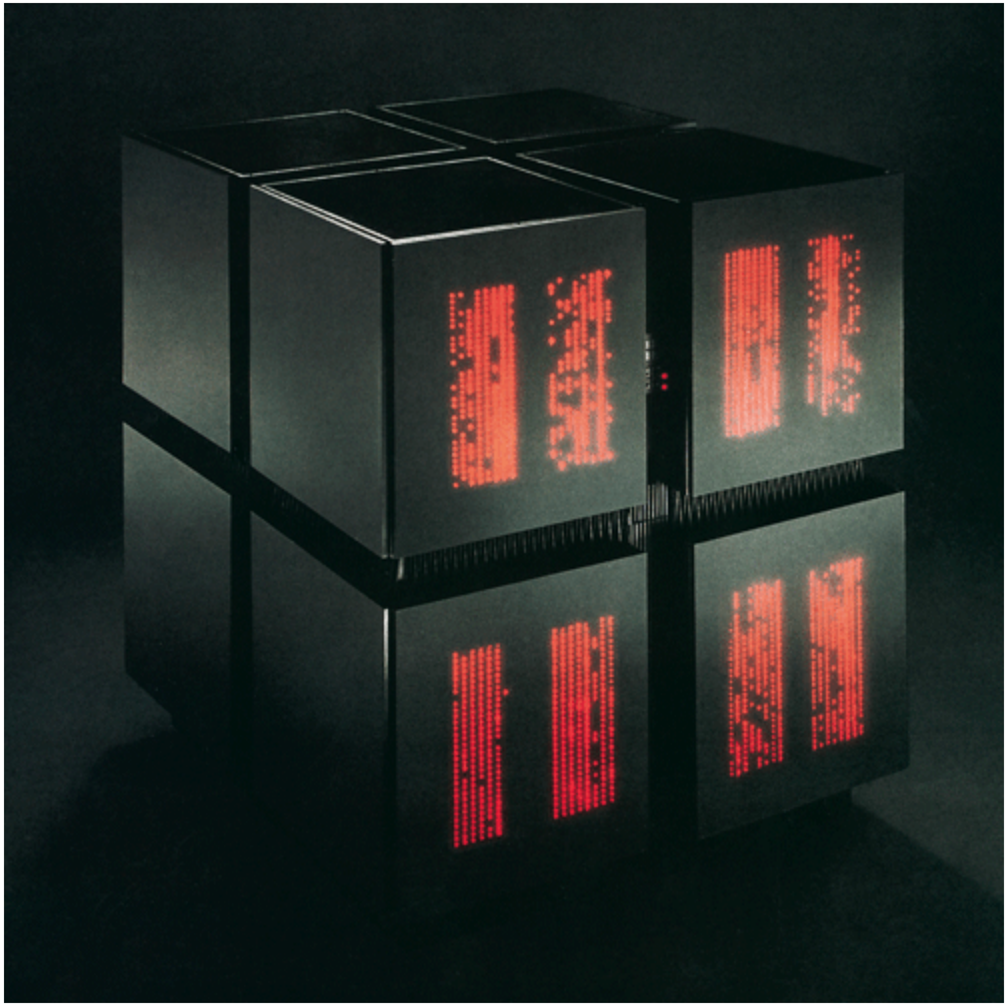
Guy E. Blelloch and John Greiner
Carnegie Mellon University
{blelloch,jdg}@cs.cmu.edu

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

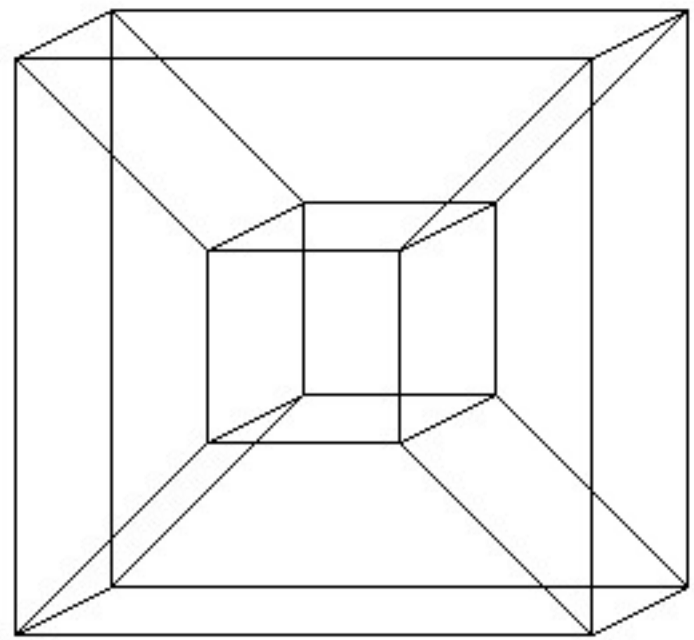
65k processors

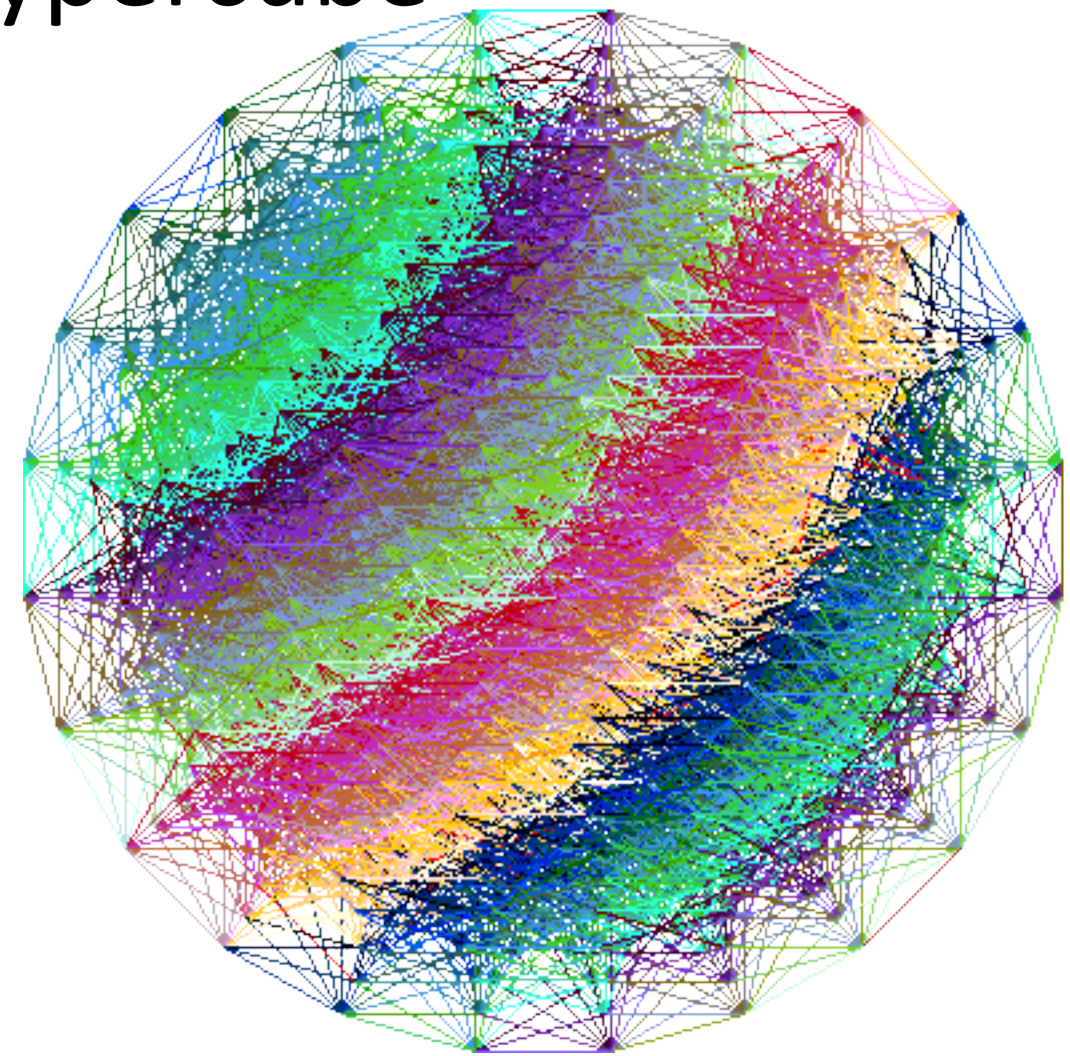
can see CM-1 and CM-5 (from 1993) at Computer History Museum, Mountain View

Image: © Thinking Machines Corporation, 1986. Photo: Steve Grohe.

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

Hypercube





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.
```

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

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

```
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};
```

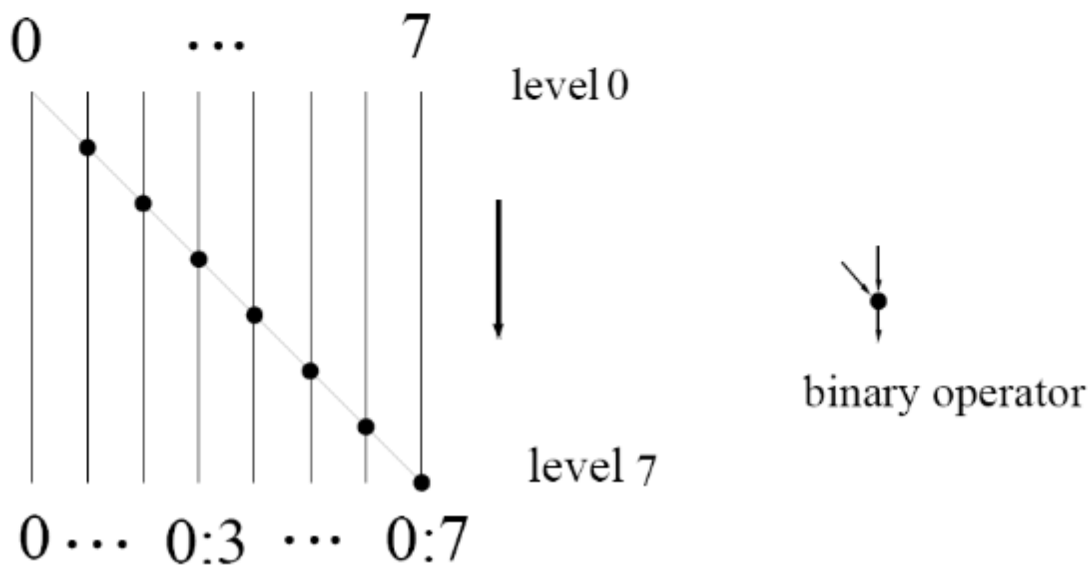
is:

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

Bye
```

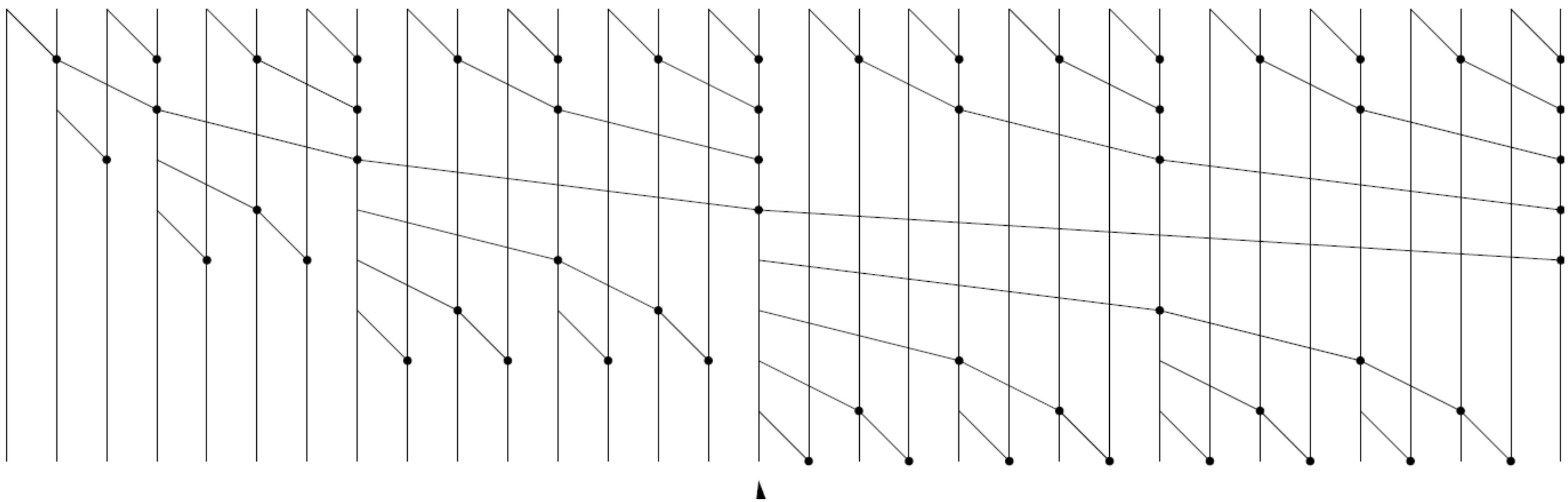
scan (Haskell first)

```
*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]
```

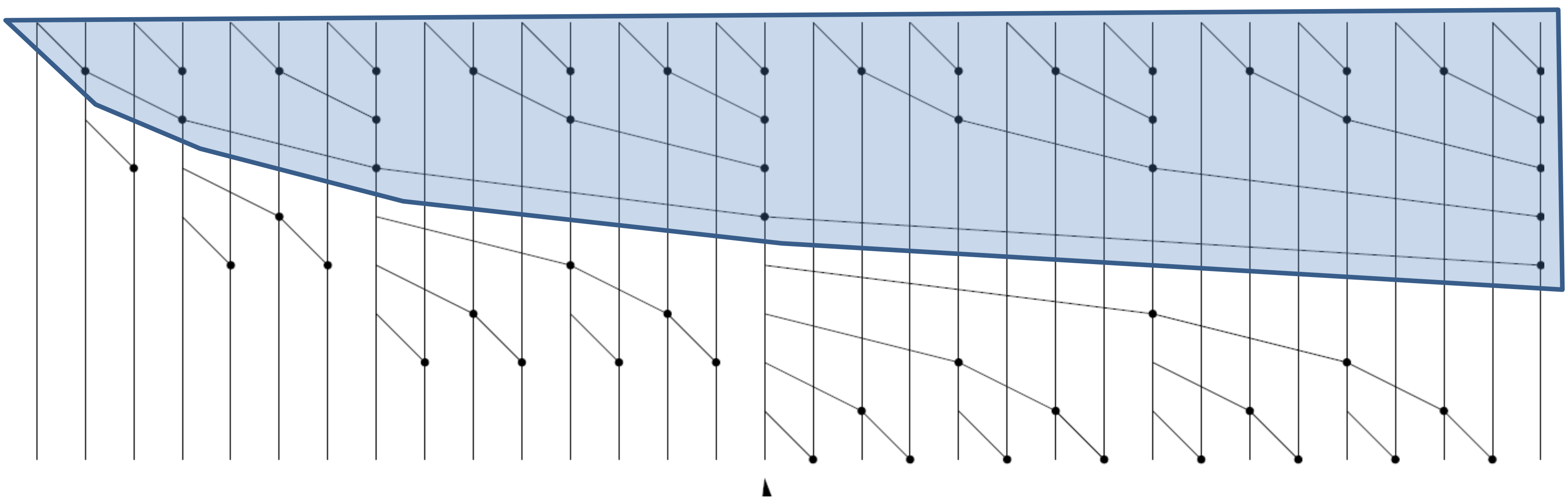
scan diagram



Brent Kung ('79)

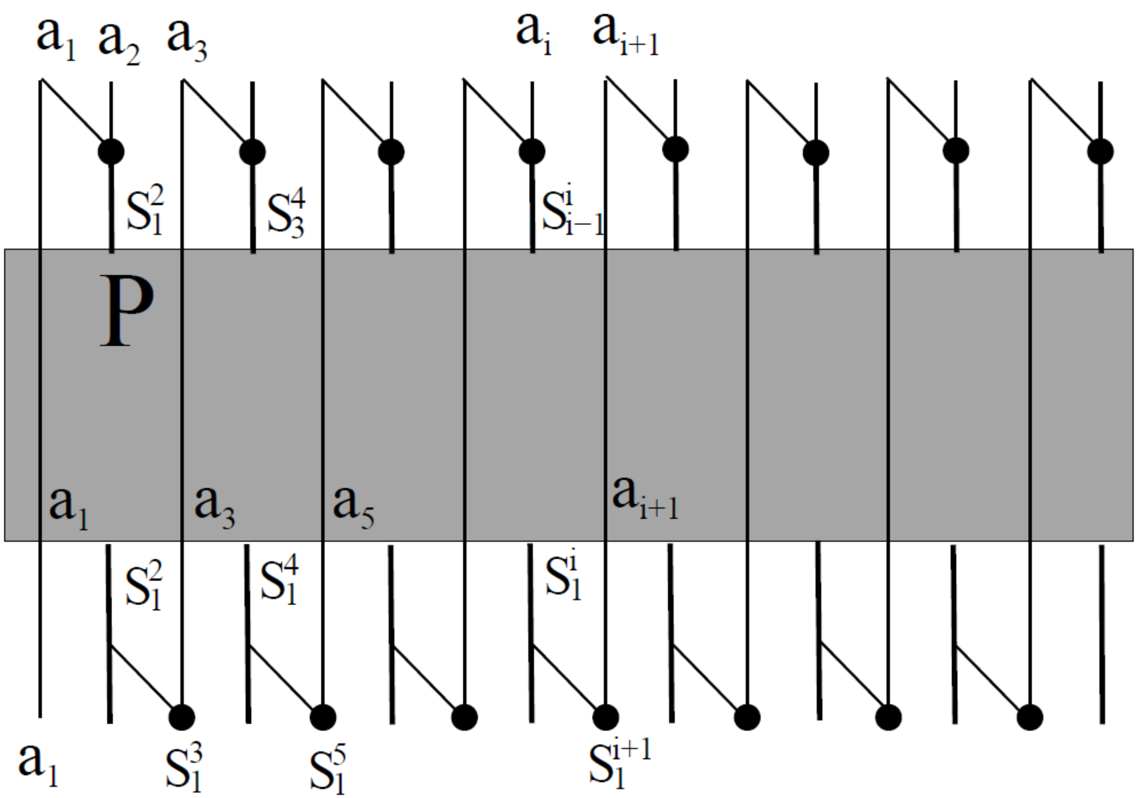


Brent Kung



forward tree + several reverse trees

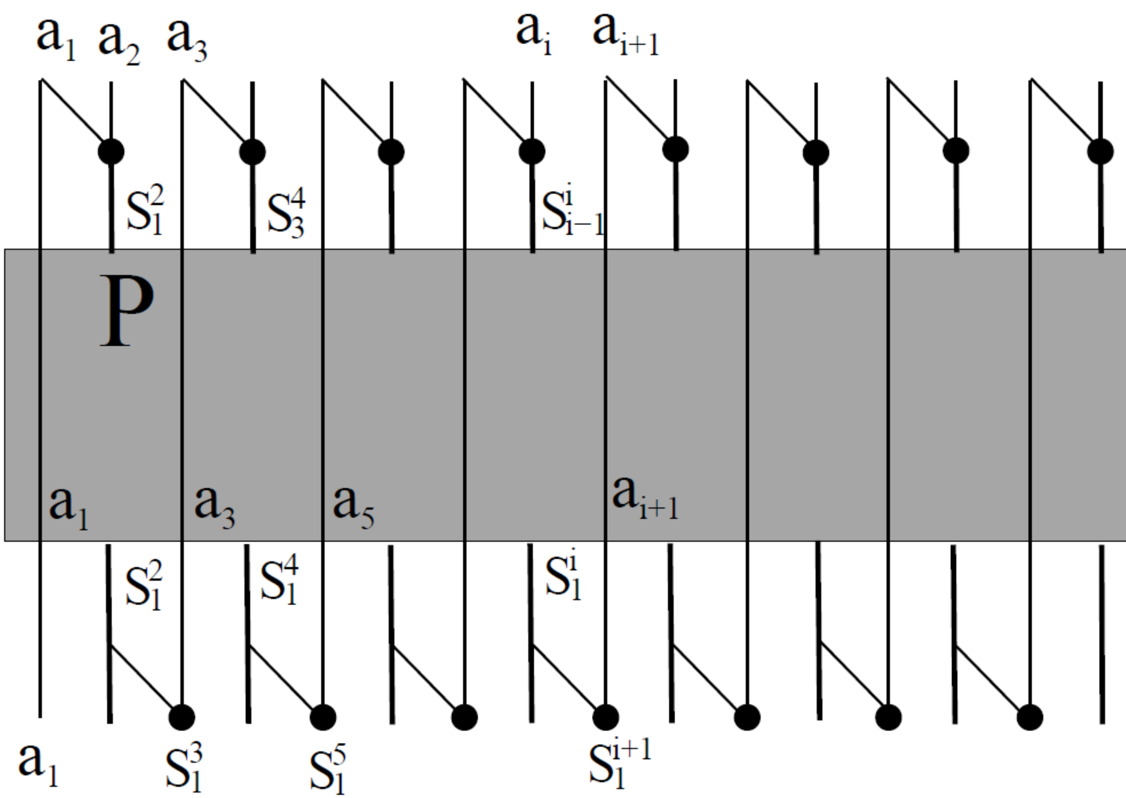
recursive decomposition



$S_i^j = a_i * a_{i+1} * \ldots * a_j$

indices from 1 here

recursive decomposition



$S_i^j = a_i * a_{i+1} * \ldots * a_j$

one recursive call on n/2 inputs

divide
conquer
combine

prescan

scan ”shifted right by one”

prescan of

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

is

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

identity element

scan from prescan

easy (constant time)

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

$*$

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

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

scan = parallel prefix

the power of scan

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

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

To evaluate polynomials

To solve recurrences. For example, to solve the recurrences

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

To implement radix sort

To implement quicksort

To solve tridiagonal linear systems

To delete marked elements from an array

To dynamically allocate processors

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

and many more

http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/ieee-scan.ps.gz

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]
```

prescan

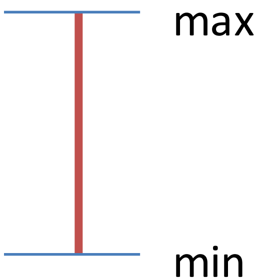
```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

```
scan_op(max, 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, 8, 8, 8, 8, 9, 9] : [int]
```

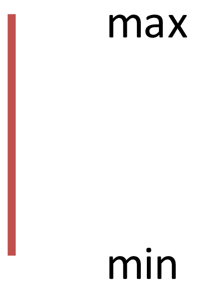
Batcher’s bitonic merge

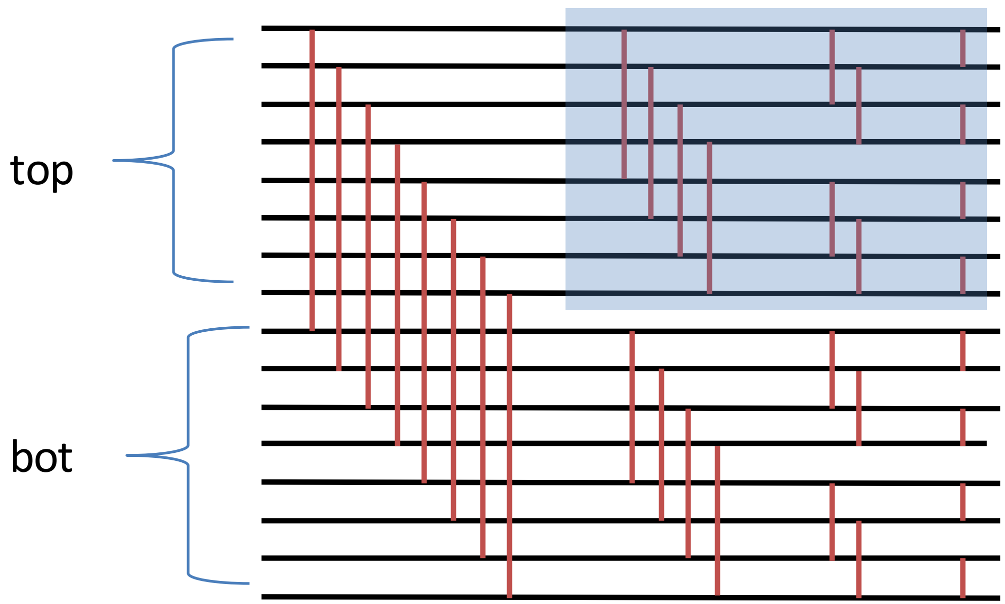
```
function bitonic_sort(a) =
if (#a == 1) then a
else
    let
            bot = subseq(a,0,#a/2);
            top = subseq(a,#a/2,#a);
            mins = {min(bot,top):bot;top};
            maxs = {max(bot,top):bot;top};
    in flatten({bitonic_sort(x) : x in [mins,maxs]});
```

bitonic_sort (merger)



bitonic_sort (merger)





bitonic sequence

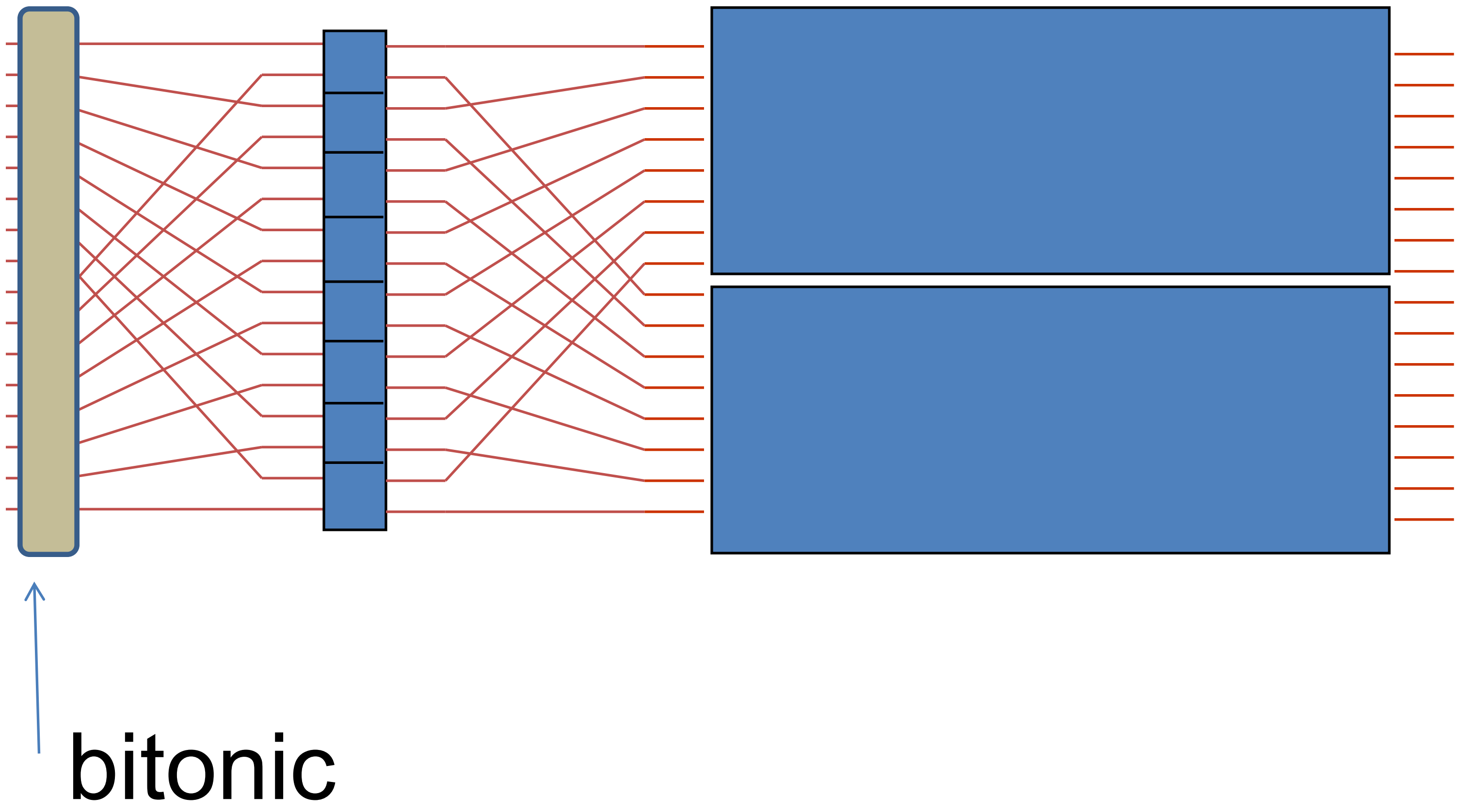
inc (not decreasing)

then

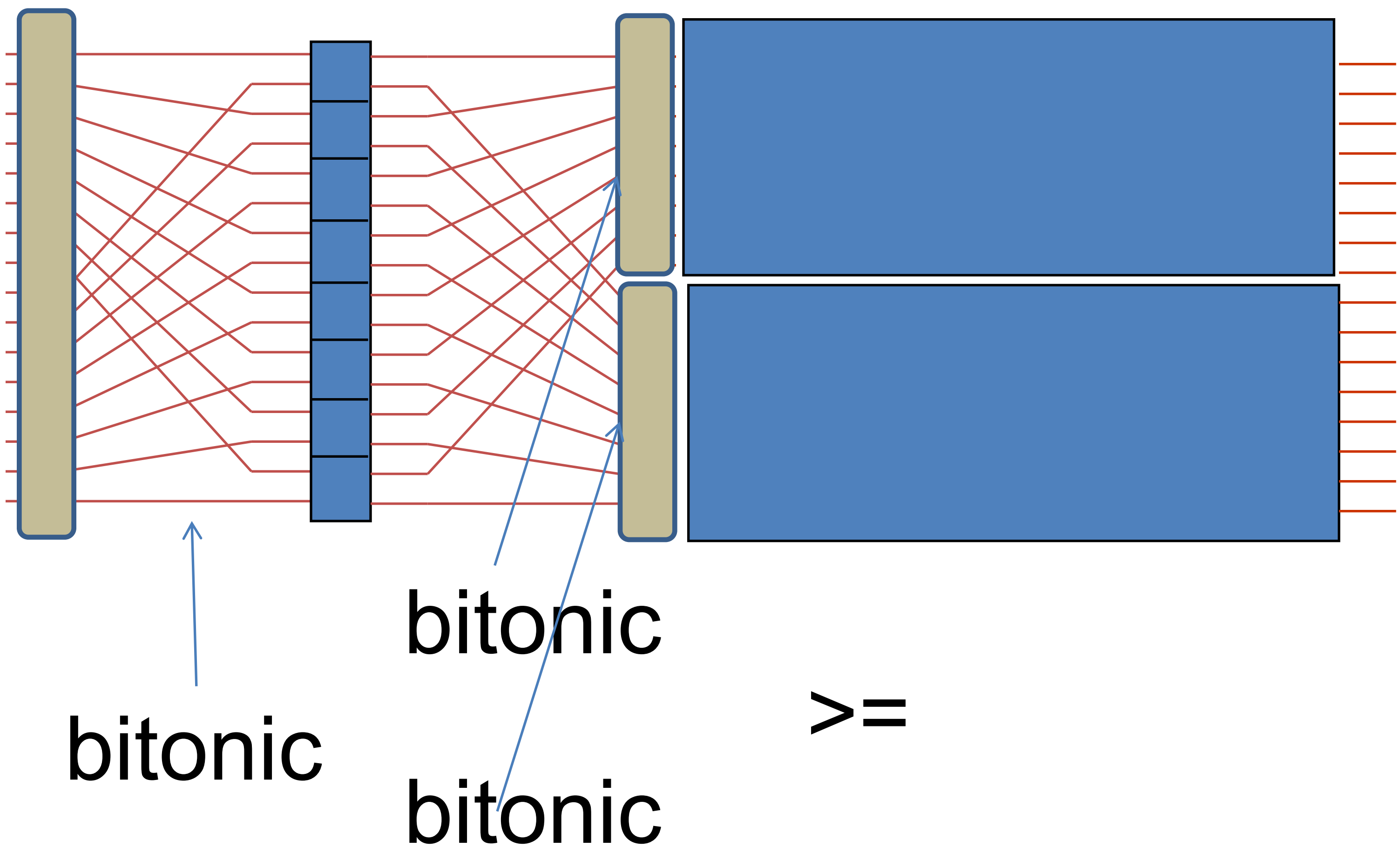
dec (not increasing)

or a cyclic shift of such a sequence

Butterfly



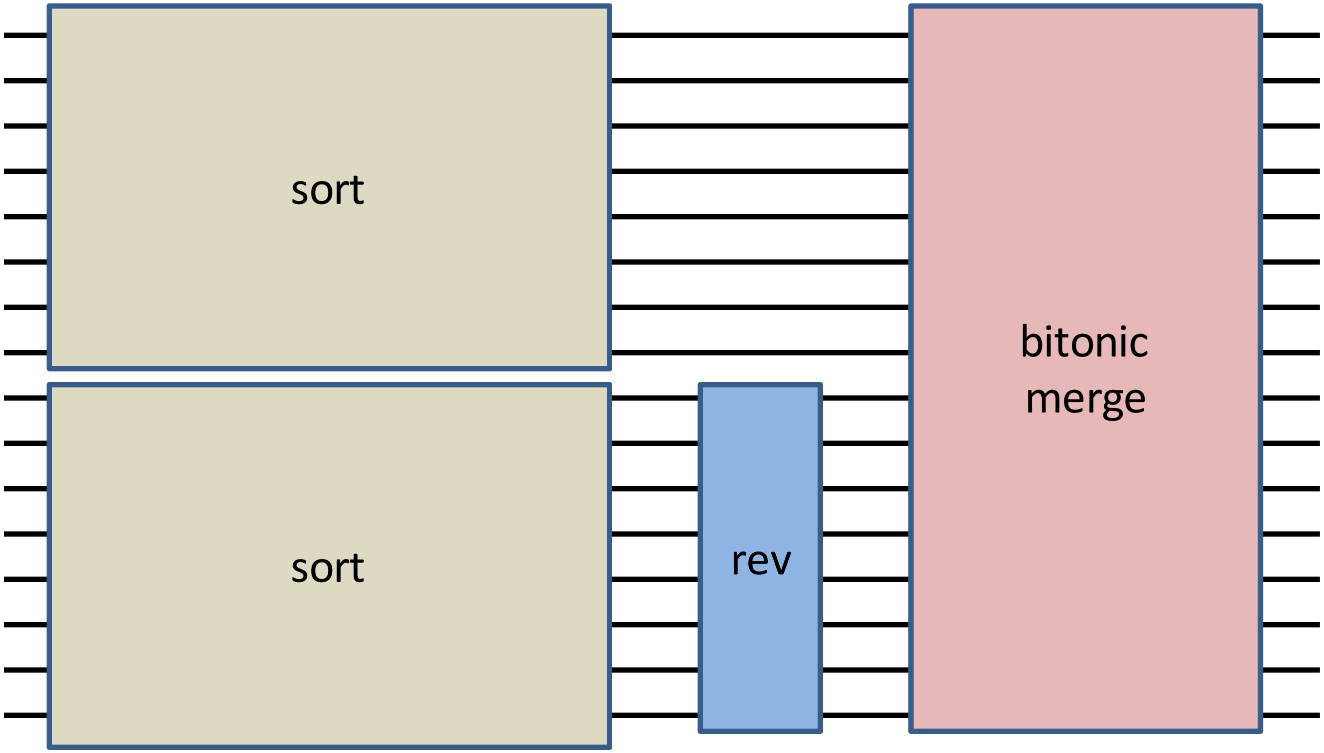
Butterfly



Now use Divide and Conquer (again) to do sorting

How??

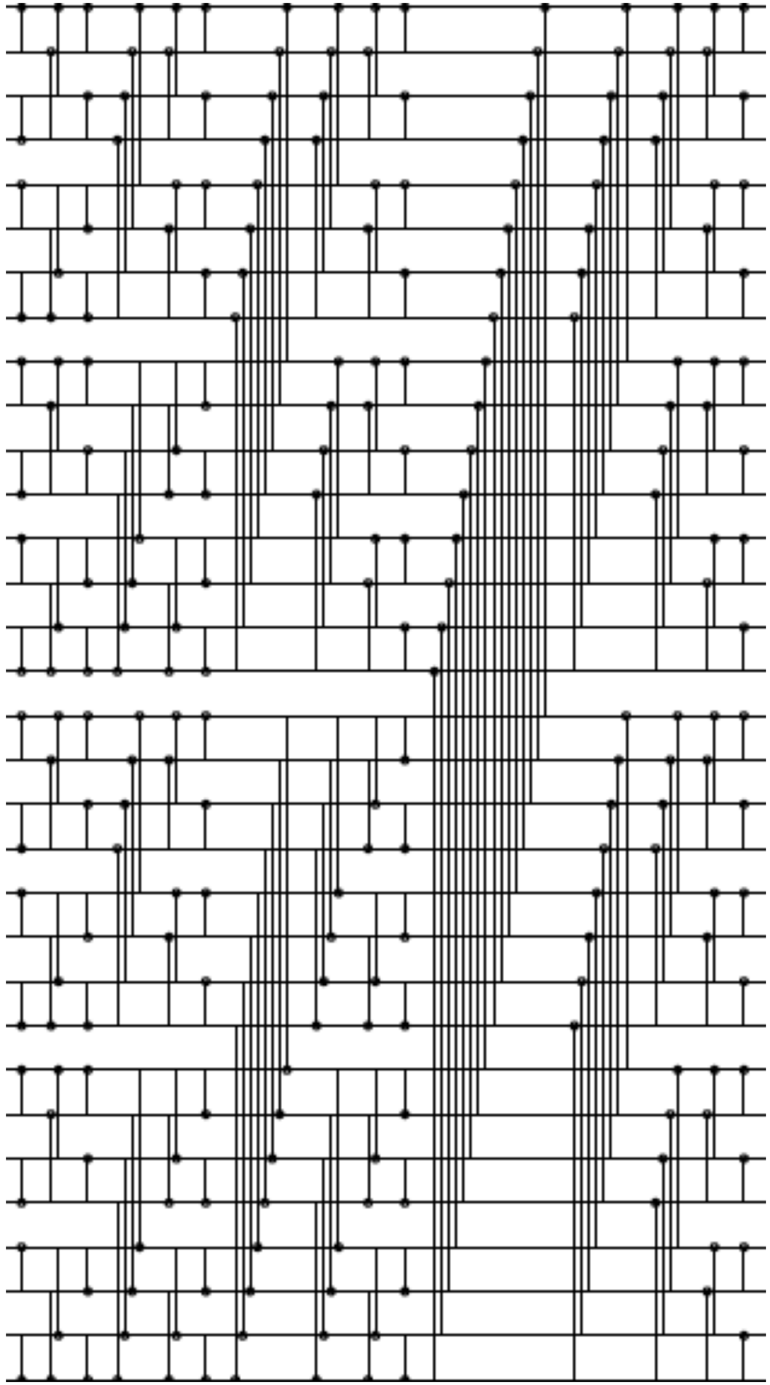
bitonic sort

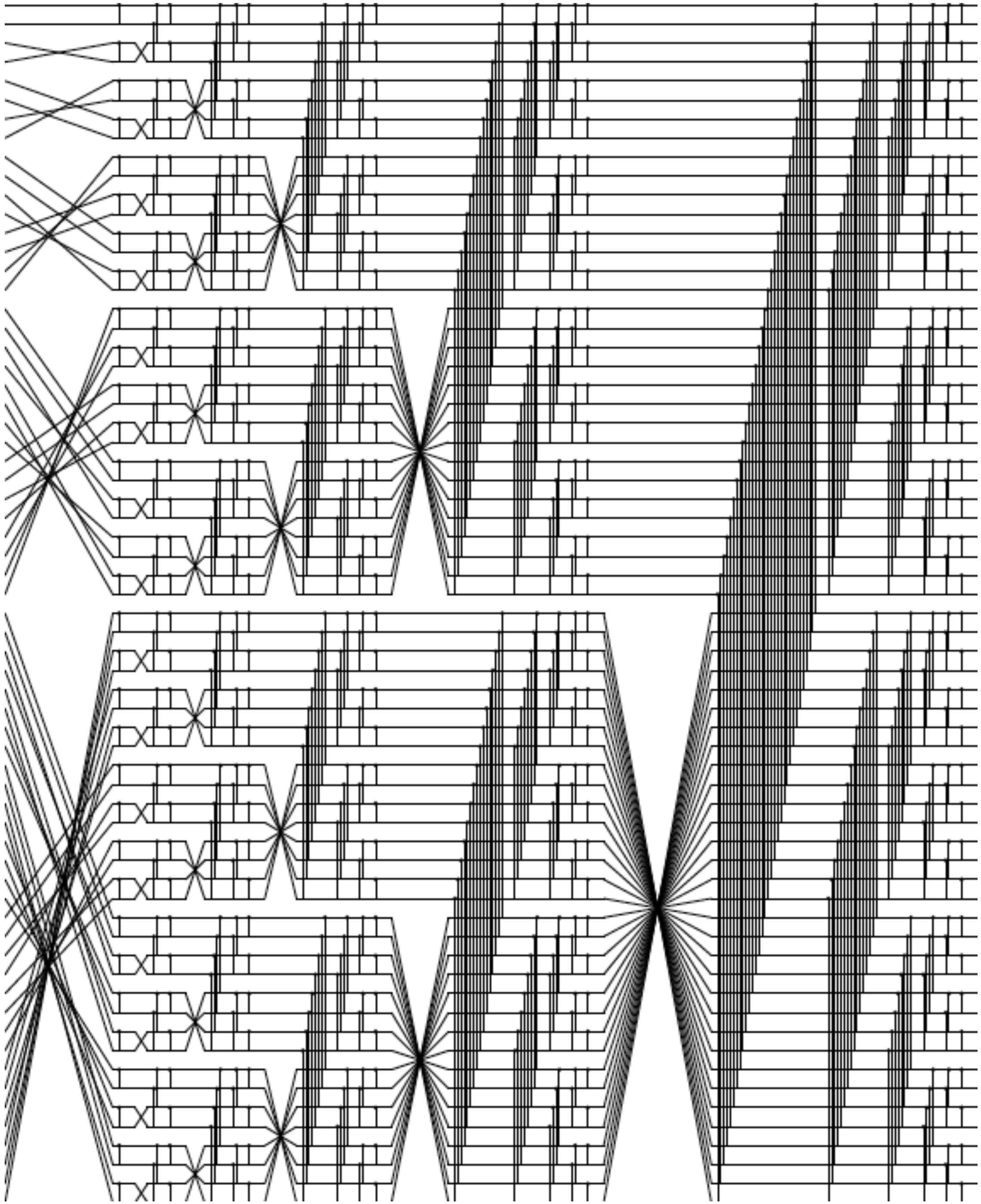


http://www.cs.kent.edu/~batcher/sort.pdf

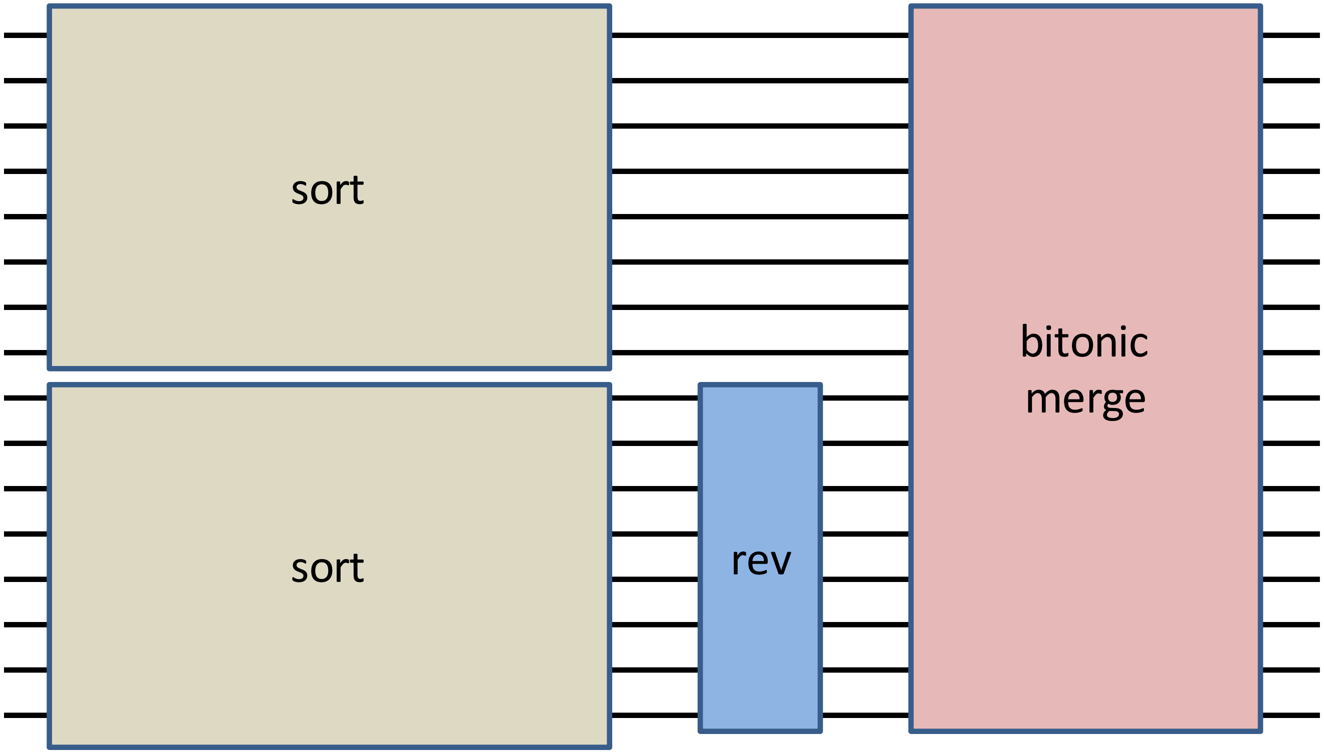
bitonic sort

```
function batcher_sort(a) =
if (#a == 1) then a
else
    let b = {batcher_sort(x) : x in bottop(a)};
    in bitonic_sort(b[0]++reverse(b[1]));
```





bitonic sort



http://www.cs.kent.edu/~batcher/sort.pdf

bitonic sort

Read Batcher's paper from 1968
It is a classic! (2753 citations on GS)

http://www.cs.kent.edu/~batcher/sort.pdf

Quicksort

```
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];
```

parentheses matching

For each index, return the index of the matching parenthesis

```
function parentheses_match(string) =
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk),  even_elts(rnk))
in permute(ret, rnk);
```

one scan, a map, a zipWith, two permutes and an interleave,
also rank and odd_elts and even_elts

parentheses matching

For each index, return the in

```
function parentheses_match(
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk),  even_elts(rnk))
in permute(ret, rnk);
```

```
permute([7,8,9],[2,1,0]);
permute([7,8,9],[1,2,0]);

it = [9, 8, 7] : [int]

it = [9, 7, 8] : [int]
```

one scan, a map, a zipWith, two permutes and an interleave, also rank and odd_elts and even_elts

parentheses matching

rank([6,8,9,7]);

it = [0, 2, 3, 1] : [int]

rank([6,8,9,7,9]);

it = [0, 2, 3, 1, 4] : [int]

For each index, return the in

```
function parentheses_match(
let
   depth = plus_scan({if c==`(
   depth = {d + (if c==`( then 1 e
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk),  even_elts(rnk))
in permute(ret, rnk);
```

one scan, a map, a zipWith, two permutes and an interleave,
also rank and odd_elts and even_elts

parentheses matching

A "step through" of this function is provided at end of these slides

For each index, return the index of the matching pa

```
function parentheses_match(string) =
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk), even_elts(rnk))
in permute(ret, rnk);
```

one scan, a map, a zipWith, two permutes and an interleave,
also rank and odd_elts and even_elts

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 }
```

Nested Parallelism

Arbitrarily nested parallel loops + fork-join

Assumes no synchronization among parallel
tasks except at join points => a task can only sync with its parent (sometimes called fully strict)

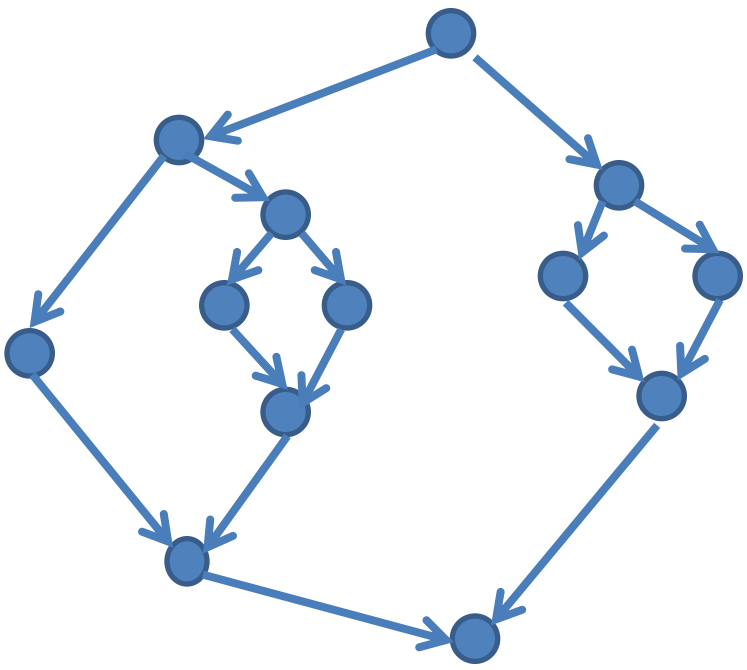
Deterministic (in absence of race conditions)

Advantages:

Good schedulers are known
Easy to understand, debug, and analyze

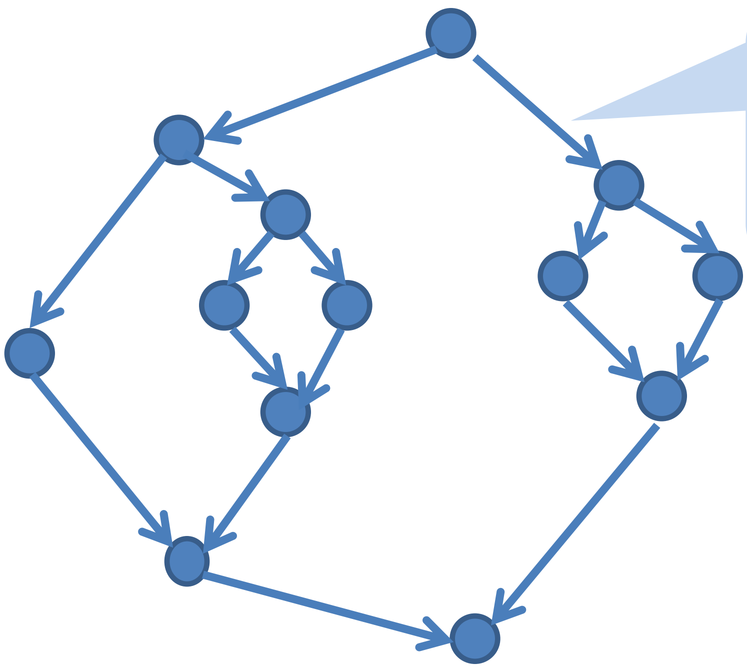
Nested Parallelism

Dependence graph is series-parallel



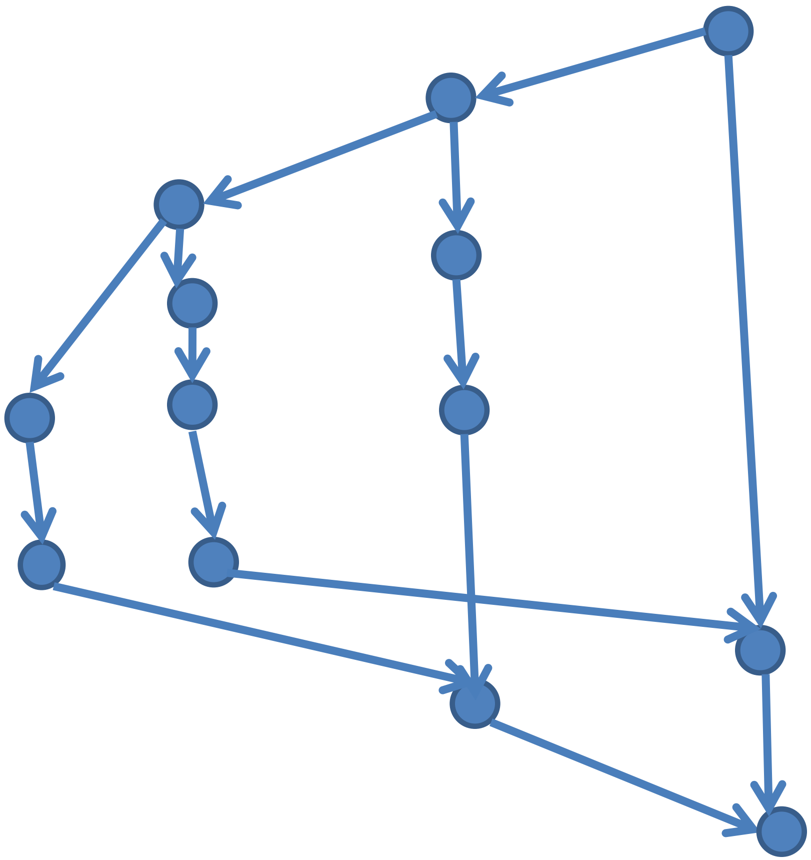
Nested Parallelism

Dependence graph is series-parallel

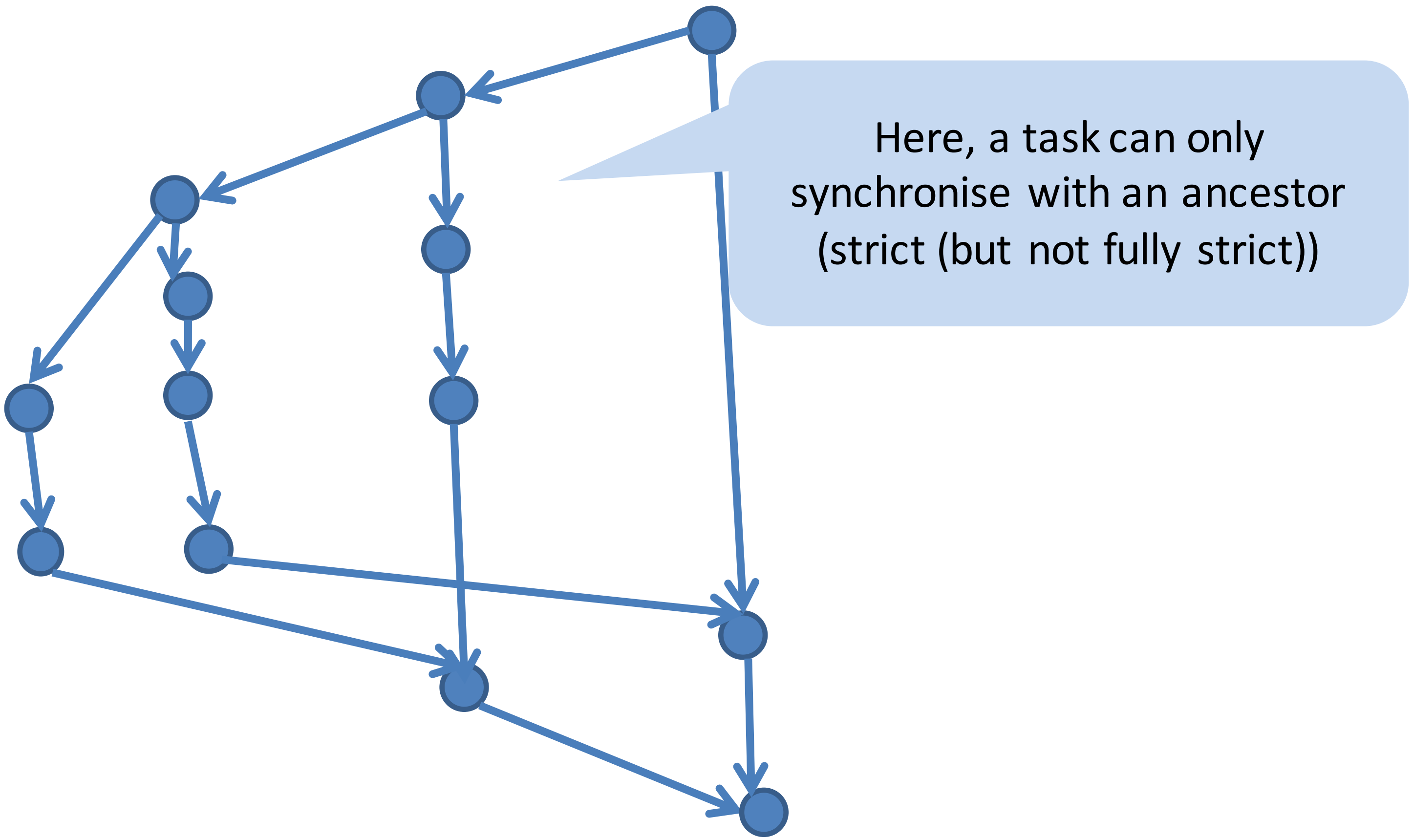


Task can only synchronise with its parent

But not



But not


Here, a task can only synchronise with an ancestor (strict (but not fully strict))

Back to examples

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});
```

Back to examples
Batcher's bitonic merge IS NESTED

```
function bitonic_sort(a) =
if (#a == 1) then a
else
   let
            bot = subseq(a,0,#a/2);
            top = subseq(a,#a/2,#a);
            mins = {min(bot,top):bot;top};
            maxs = {max(bot,top):bot;top};
   in flatten({bitonic_sort(x) : x in [mins,maxs]});
```

and so is the sort

Back to examples
Batcher's bitonic merge IS NESTED

nestedness is good for D&C
and for irregular computations

```
function bitonic_sort(a) =
if (#a == 1) then a
else
    let
            bot = subseq(a,0,#a/2);
            top = subseq(a,#a/2,#a);
            mins = {min(bot,top):bot;top};
            maxs = {max(bot,top):bot;top};
    in flatten({bitonic_sort(x) : x in [mins,maxs]});
```

and so is the sort

Back to examples parentheses matching is FLAT

For each index, return the index of the matching parenthesis

```
function parentheses_match(string) =
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk),  even_elts(rnk))
in permute(ret, rnk);
```

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

Work w

on a sequential machine = sequential time

but can maybe be shared among multiple processors

Evenly shared work on #proc processors would take (about) w/#proc time

Work w

on a sequential machine = sequential time

but can maybe be shared among multiple processors

Evenly shared work on #proc processors would take (about) w/#proc time   perfect speedup

Span s

(or depth)

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

Span s

(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

scheduler

Assume a “reasonable” 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

scheduler

Assume a “reasonable” 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

Then runtime <= (work / #proc) + span

runtime <= (work / #proc) + span

If the first term dominates, then we are getting near perfect speedup (within a factor of 2)

Define

Parallelism = work / span

Number of processors for which the two terms are equal

Gives rough upper bound on number of processors can use effectively

Part 1: simple language based performance model

Call-by-value λ-calculus

$$\lambda x.\, e \Downarrow \lambda x.\, e \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\, e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v'}{e_1\, e_2 \Downarrow v'} \quad \text{(APP)}$$

slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus: cost model

$$e \Downarrow\ v;\, w,d$$

Reads: expression *e* evaluates to *v* with work *w* and span *d.*

- **Work** (W): sequential work
- **Span** (D): parallel depth

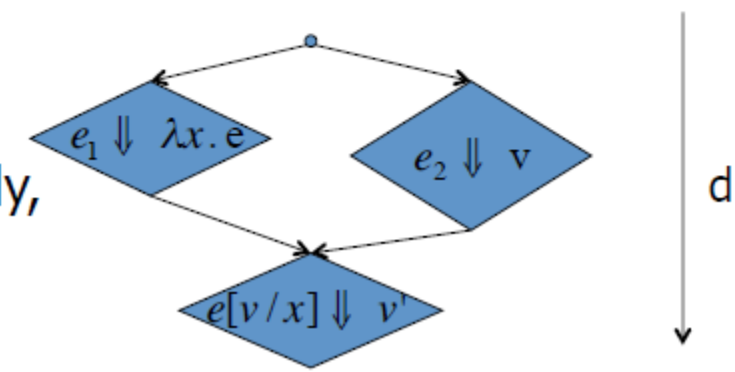
slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus: cost model

$$\lambda x.\mathrm{e} \Downarrow \lambda x.\mathrm{e};\ \boxed{1},\boxed{1} \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.e;\ \boxed{w_1},\boxed{d_1} \quad e_2 \Downarrow v;\ \boxed{w_2},\boxed{d_2} \quad e[v/x] \Downarrow v';\ \boxed{w_3},\boxed{d_3}}{e_1\, e_2 \Downarrow v';\ \boxed{1 + w_1 + w_2 + w_3},\ \boxed{1 + \max(d_1, d_2) + d_3}} \qquad \text{(APP)}$$

Work adds
Span adds sequentially,
and max in parallel



slide from Blelloch’s ICFP10 invited talk

The Parallel λ-calculus cost model

$$\lambda x.\mathrm{e} \Downarrow \lambda x.\mathrm{e};\ 1,1 \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\,e;\ w_1,d_1 \quad e_2 \Downarrow v;\ w_2,d_2 \quad e[v/x] \Downarrow v';\ w_3,d_3}{e_1\,e_2 \Downarrow v';\ 1+w_1+w_2+w_3,\ 1+\max(d_1,d_2)+d_3} \qquad \text{(APP)}$$

$$c \Downarrow c;\ 1,1 \qquad \text{(CONST)}$$

$$\frac{e_1 \Downarrow c;\ w_1,d_1 \quad e_2 \Downarrow v;\ w_2,d_2 \quad \delta(c,v) \Downarrow v'}{e_1\,e_2 \Downarrow v';\ 1+w_1+w_2,\ 1+\max(d_1,d_2)} \qquad \text{(APPC)}$$

$$c_n = 0,\cdots,n,+,+_0,\cdots,+_{\mathrm{n}},<,<_0,\cdots,<_{\mathrm{n}},\times,\times_0,\cdots,\times_{\mathrm{n}},\cdots \qquad \text{(constants)}$$

slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

$$\frac{e'[v_i / x] \Downarrow v_i';\, w_i, d_i \qquad i \in \{1 \ldots n\}}{\{e' : x \text{ in } [v_1 \ldots v_n]\} \Downarrow [v_1' \ldots v_n'];\; 1 + \sum_{i=1}^{n} w_i,\; 1 + \max_{i=1}^{|v|} d_i}$$

Primitives:

```
<- : 'a seq * (int,'a) seq -> 'a seq
```

- `[g,c,a,p] <- [(0,d),(2,f),(0,i)]`
  `[i,c,f,p]`

[ICFP95]

```
elt, index, length
```

slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

Blelloch:
programming based cost models could change the way people think about costs and open door for other kinds of abstract costs
doing it in terms of machines.... "that's so last century"

```
<- : 'a seq * (int,'a) seq -> 'a seq
• [g,c,a,p] <- [(0,d),(2,f),(0,i)]
      [i,c,f,p]

elt, index, length
```

[ICFP95]

slide from Blelloch's ICFP10 invited talk

The Second Half: Provable Implementation Bounds

Theorem [FPCA95]: If $e \Downarrow v; w, d$ then v can be calculated from e on a CREW PRAM with p processors in $O\left(\frac{w}{p} + d \log p\right)$ time.

Can't really do better than: $\max\left(\frac{w}{p}, d\right)$

If w/p > d log p then "work dominates"

We refer to w/d as the parallelism.

(Typo fixed by MS based on the video)

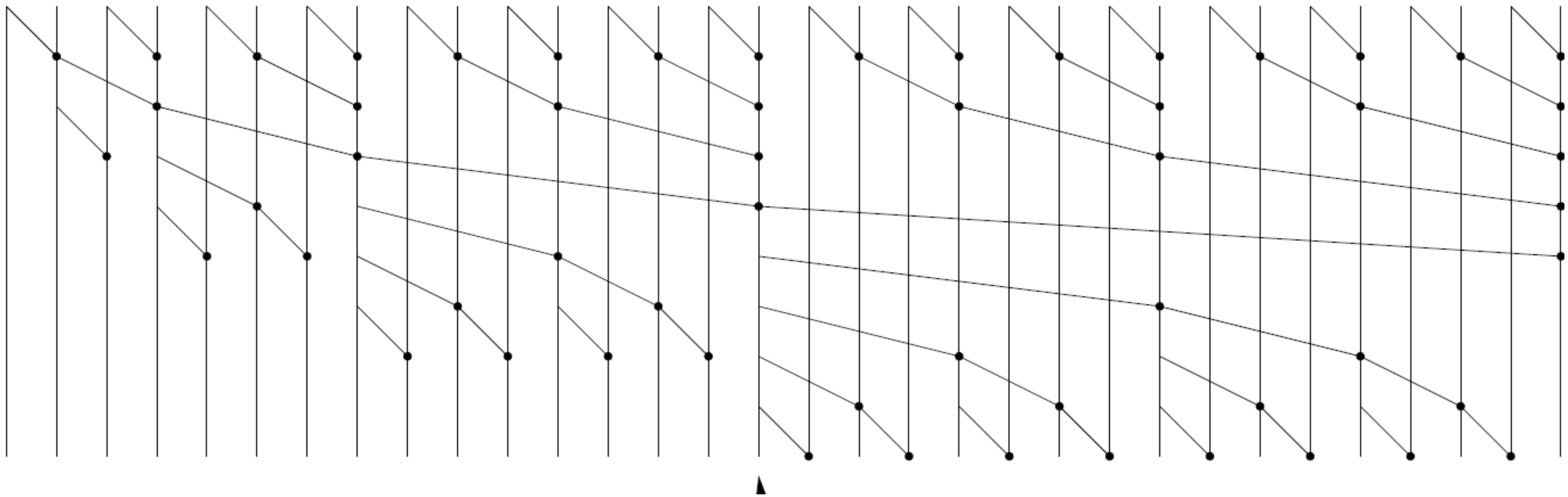
slide from Blelloch's ICFP10 invited talk

Brent’s lemma

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 $t + (q-t)/p$ steps with p processors

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

Back to our scan



oblivious or data independent computation

$N = 2^n$ inputs, work of dot is 1
work = ?
depth = ?

and bitonic sort?

Quicksort

```
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];
```

Analysis in ICFP10 video gives   depth = O(log N)   work = O(N logN)

Quicksort

```
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];
```

Analysis in ICFP10 video gives depth = O(log N) work = O(N logN)

(The depth is improved over the example with trees, due to the addition of parallel arrays as primitive.)

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 : 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

Data Parallel Haskell (DPH) intentions

NESL was a seminal breakthrough but, fifteen years later it remains largely un-exploited. Our goal is to adopt the key insights of NESL, embody them in a modern, widely-used functional programming language, namely Haskell, and implement them in a state-of-the-art Haskell compiler (GHC). The resulting system, Data Parallel Haskell, will make nested data parallelism available to real users.

Doing so is not straightforward. NESL a first-order language, has very few data types, was focused entirely on nested data parallelism, and its implementation is an interpreter. Haskell is a higher-order language with an extremely rich type system; it already includes several other sorts of parallel execution; and its implementation is a compiler.

http://www.cse.unsw.edu.au/~chak/papers/fsttcs2008.pdf

NESL also influenced

The Java 8 streams that you saw on Monday!!

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

Collections seems 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).

End

parentheses matching

For each index, return the index of the matching parenthesis

```
function parentheses_match(string) =
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk), even_elts(rnk))
in permute(ret, rnk);
```

(	)	(	(	)	(	)	)	(	(	(	)	)	)
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1

( ) (( ) ( )) ((( )))

1 -1 1 1 -1 1 -1 -1 1 1 1 -1 -1 -1

0 1 0 1 2 1 2 1 0 1 2 3 2 1

prescan
(+)

(	)	(	(	)	(	)	)	(	(	(	)	)	)
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1
0	1	0	1	2	1	2	1	0	1	2	3	2	1
1	1	1	2	2	2	2	1	1	2	3	3	2	1

+1 if (
+0 if)

( ) (( ) ( )) ((( )))

1 -1 1 1 -1 1 -1 -1 1 1 1 -1 -1 -1

0 1 0 1 2 1 2 1 0 1 2 3 2 1

+1 if (
+0 if)

1 1 1 2 2 2 2 1 1 2 3 3 2 1 depth

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	7	8	13	3	4	5	6	9	12			

permute
([0:#string),rank(depth));

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk
1	0	7	2	13	8	4	3	6	5	2	9	11	10	ret

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk
1	0	7	2	13	8	4	3	6						

interleave(odd_elts(rnk), even_elts(rnk))

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
1	0	7	2	13	8	4	3	6	5	2	9	11	10	ret
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk
1	0	7	4	3	6	5	2	13	12	11	10	9	8	

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
1	0	7	2	13	8	4	3	6	5	2	9	11	10	ret
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk
1	0	7	4	3	6	5	2	13	12	11	10			

permute(ret,rnk);

( ) (( ) ( )) ((( ))) string

1 0 7 4 3 6 5 2 13 12 11 10 9 8