Data Parallel Languages

The First?

APL

("A Programming Language")

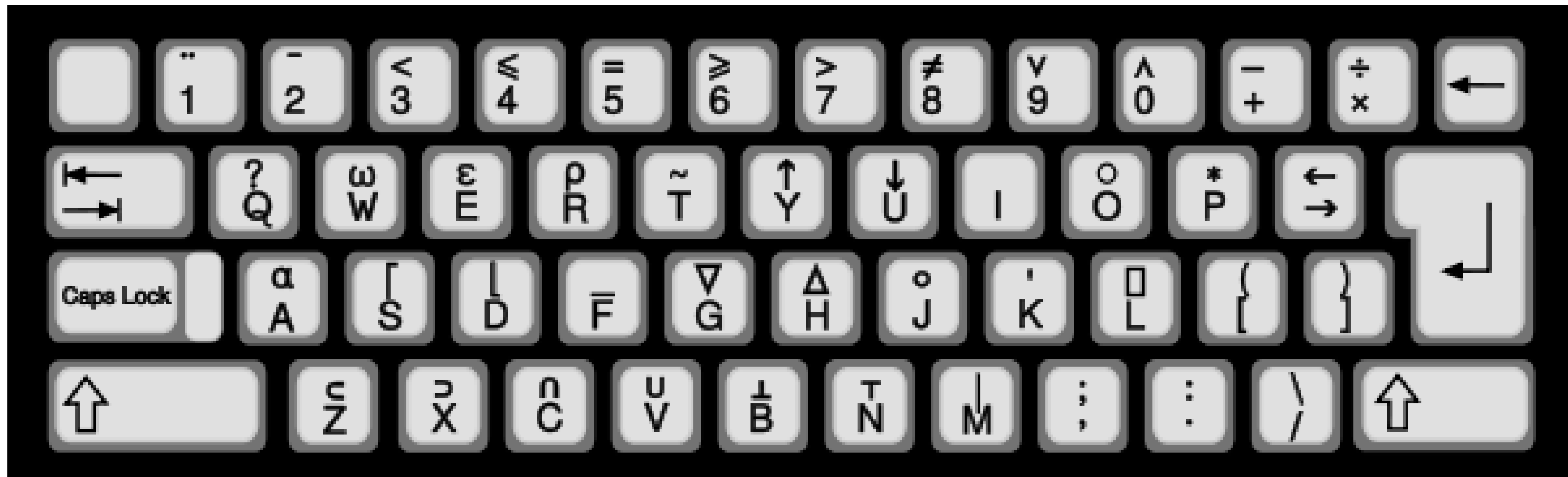
Kenneth Iverson

APL Idea

- Iverson's thesis, Harvard, 1957
- Book 1965
- IBM Interpreters, late 60's
- Among earliest *interactive* (calculator-like) systems
- Specialized keyboards, font set
- Focus
  - **Succinct** notation (one-liners)
  - Optimizations
- Parallel machines were pretty rare then!
- Later language: "J"

APL Keyboard & Samples

(source: http://en.wikipedia.org/wiki/APL_(programming_language))



Prime Numbers from 1 to R

- The following expression finds all prime numbers from 1 to R. In both time and space, the calculation is O(R²):

  (~R∊R∘.×R)/R←1↓ιR
- Executed from **right to left**, this means:
  - ιR creates a vector containing integers from 1 to R (if R = 6 at the beginning of the program, ιR is 1 2 3 4 5 6)
  - Drop first element of this vector (↓ function), i.e. 1. So 1↓ιR is 2 3 4 5 6
  - Set R to the new vector (←, assignment primitive), i.e. 2 3 4 5 6
  - Generate outer product of R multiplied by R, i.e. a matrix which is the multiplication table of R by R (°.× function), i.e.

4	6	8	10	12
6	9	12	15	18
8	12	16	20	24
10	15	20	25	30
12	18	24	30	36

  - Build a vector the same length as R with 1 in each place where the corresponding number in R is in the outer product matrix (∊, set inclusion function), i.e. 0 0 1 0 1
  - Logically negate the values in the vector (change zeros to ones and ones to zeros) (~, negation function), i.e. 1 1 0 1 0
  - Select the items in R for which the corresponding element is 1 (/ function), i.e. 2 3 5

Game of Life

Michael Gertelman has coded Conway's Game of Life in one line of APL:

```
⍎'⎕',∊Nρ⊂S←'←⎕←(3=T)∨M∧2=T←⊃+/(V⌽''⊂M),(V⊖''⊂M),(V,⌽V)⌽''(V,V←1 ¯1)⊖''⊂M'
```

http://www.youtube.com/watch?gl=GB&hl=en-GB&v=a9xAKttWgP4&fmt=18

Fortran 90, HPF (High-Performance Fortran)

Fortran-90, a superset of Fortran-77, is an ISO and ANSI standard language, with extensions that include facilities for array operations. In Fortran-90, a statement such as

```
A = SQRT(A) + B ** 2
```

squares every element of array B, extracts the square root of every element of array A, and adds the corresponding elements of the two arrays, storing the results in array A. As a second example,

```
WHERE (B /= 0) A = A / B
```

performs a masked array assignment, resulting in each element of A being divided by the corresponding element of B, except in those cases where the B element is 0.

The semantics of Fortran-90 is independent of the underlying machine model. It simply provides a global name space and a single thread of control. However, array operations of the type presented above allow Fortran-90 programs to be efficiently executed on parallel machines. When run on a distributed-memory machine, some Fortran-90 constructs imply interprocessor communication. Assignment of a scalar value to an array

```
A = S/2
```

may imply multicasting or broadcasting (one-to-many communication). Use of "array section" notation or index vectors

```
A(I:J) = B(J:I:-1)   {a section of array B is assigned, in reverse order, to array A}
A(P) = B             {P is an integer index vector; means A(P(I)) = B(I) for all I}
```

may require data permutation (many-to-many communication). Finally, reduction operations, such as summing all elements of an array

```
S = SUM(B)
```

may require a gather operation (many-to-one communication).

Vector Processors

- These are high-performance architectures for processing vectors.
- Early ones were by Cray, Fujitsu, NEC.
- Parallelism within vector pipeline.
- Multiple pipelines on some machines.
- The idea resurfaces every now and then.

Matlab

- Matlab follows a strategy similar to Fortran.
- A parallel version of Matlab now exists.

Thinking Machines

- *Lisp
- *C
- Star-P
  applied to Matlab and NumPy
  http://star-p.org/products/

Database Query Languages

- Query languages, such as SQL, are essentially special-purpose data-parallel languages.
- Hadoop distributed database uses map/reduce for parallel execution.

NESL: “Nested Parallelism Language”

- Guy Blelloch @ CMU (dissertation from MIT)
- A language coupled with a parallel complexity theory
- Functional, data-parallel, borrowing from APL, SETL, ML, Miranda, Sisal, …
- Strongly-typed!
- Implemented on a variety of parallel machines (CM-2, 5, Cray C90, MPI, …)
- Lightweight sequential implementation in Lisp
- Web-based CGI evaluator
- “Mostly dormant since 1997” (from 2006 talk)

- Concise specification of parallel algorithms, whether or not you use it for execution

Basis for Complexity

- Organizing by vectors makes counting easier.
- VRAM: Vector Random-Access Machine (not Video RAM!)
- Similar to PRAM, but with **vector instructions**.
- Assumes **scan** (= parallel prefix) operations can be done fast, O(log n) depth, O(n) work for length n vectors.
- Assume vector operations such as **map** are O(1).

Work/Depth

- Algorithms analyzed by work and depth, similar to work/span in Leiserson’s analysis (which came later).

- Inspired by earlier book by Joseph JaJa.

Layered Implementation

- NESL hides the CPU/Memory allocation, and inter-processor communication details by providing an abstraction of parallelism.
- The current NESL implementation is based on an intermediate language (VCODE)and a library of low level vector routines (CVL).
- **"Implementation of a Portable Nested Data-Parallel Language"** Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha.

NESL scan primitive

- associative binary operator $\oplus$
- identity I
- elements $a_0, a_1, \ldots, a_{n-1}$
- returns
  $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

- We can get, in one additional parallel $\oplus$:
  $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$

Scan Examples

- arg vector: [3, 5, 2, 7, 6, 1, 4]
- results:
  - +-scan: [0, 3, 8, 10, 17, 23, 24]
  - max-scan [$-\infty$, 3, 5, 5, 7, 7, 7]
  - min-scan [∞, 3, 3, 2, 2, 2, 1]
  - copy: [3, 3, 3, 3, 3, 3, 3]
    (what is operator and Identity?

More Scan Examples

- arg vector: [F, T, T, F, T, F, F]
- results:
  - or-scan: [F, F, T, T, T, T, T]
  - and-scan [T, F, F, F, F, F, F]

- “enumerate” operation:
  add up the number of T’s to the left:
  enumerate => [0, 0, 1, 2, 2, 3, 3]

More Scan Examples

- arg vector [F, T, T, F, T, F, F]
- "**enumerate-x**" operation:
  add up the number of x's to the left:
  enumerate-T => [0, 0, 1, 2, 2, 3, 3]

- "**back-enumerate-x**" operation:
  add up the number of x's to the right:
  back-enumerate-T => [2, 2, 1, 1, 0, 0, 0]

permute primitive

- permute(Vector, PermutationVector)

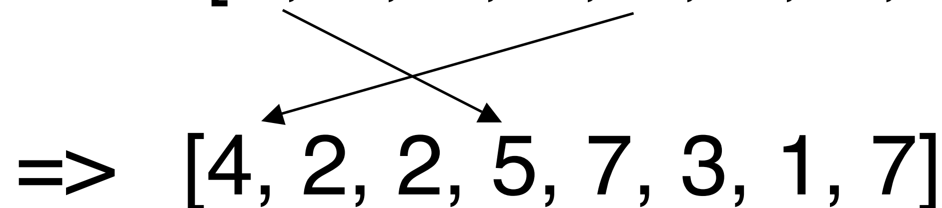
permute([3, 1, 5, 1, 2, 4],
[2, 4, 1, 0, 3, 5])

=> [1, 5, 3, 2, 1, 4]

Splitting

- Packs Vector elements corresponding to F flag in left part, T flag in right part:

split([5, 7, 3, 1, 4, 2, 7, 2],
        [T, T, T, T, F, F, T, F])



=> [4, 2, 2, 5, 7, 3, 1, 7]

Exercise

- How to implement **split** using scan operations?
  - Determine new index for each element:
    - **Enumerate** F determines indices for lower part
    - **Back-enumerate** T using complement vector determines indices for upper part
    - Compute vector of length-elements above
  - **Select** one index or the other, based upon original T-F vector
  - **Permute**
  - **About 5 VRAM steps**

Example

- split([5, 7, 3, 1, 4, 2, 6, 0],
  [T, T, T, T, F, F, T, F]):
- enumerate-F => [0, 0, 0, 0, 0, 1, 2, 2]
- back-enum-T => [4, 3, 2, 1, 1, 1, 0, 0]
- subtract back-enum from length-1 (7)
  => [3, 4, 5, 6, 6, 6, 7, 7]
- Select from one of the two vectors based on T-F => [3, 4, 5, 6, 0, 1, 7, 2]
- Permute => [4, 2, 0, 5, 6, 3, 1, 7]

Using split to Implement Radix Sort

- **Assume d-bit numbers**
- V = original vector of numbers;
  for I = 0 to d-1
  {
  Flags = i^{th} bit of numbers;
  V = split(V, Flags);
  }
- O(d) VRAM steps

Representation of Nested Lists

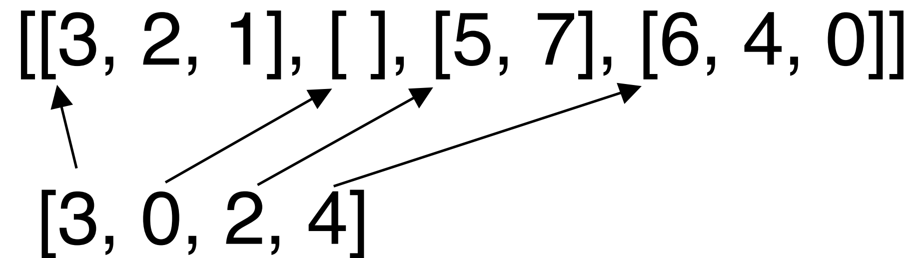
- Customarily we use pointer structures
- Instead, NESL / VRAM implementation [sometimes?] uses a **bit vector** to represent **segment boundaries**:
- Example: The **head-flags method**

  [[3, 2, 1], [5, 7], [6, 4, 0]]

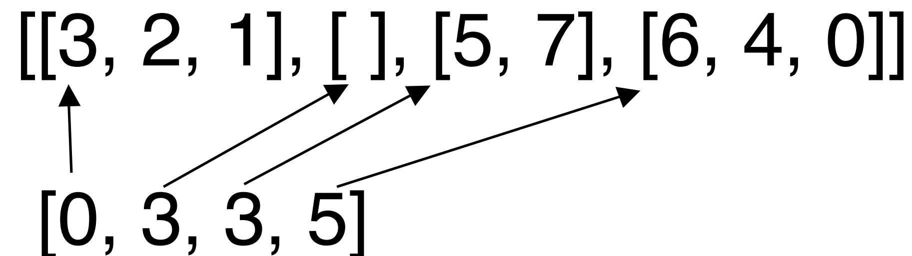
  [T, F, F, T, F, T, F, F]
- This method cannot represent **empty** segments however.

Representation of Nested Lists

- Example: The **lengths representation**



- Example: The **head-pointers representation**



Segmented scan operations

- These are scan operations done **separately within each segment.**
- Example with **head-flags representation**

  [3, 2, 1, 5, 7, 6, 4, 0]

  [T, F, F,T, F, T, F, F]
- **seg-+-scan** =>

  [0, 3, 5, 0, 5, 0, 6, 10]

Segmented scan operations

- Example with **head-flags method**

  [3, 2, 1, 5, 7, 6, 4, 0]

  [T, F, F, T, F, T, F, F]

- seg-+-scan =>

  [0, 3, 5, 0, 5, 0, 6, 10]

Flatten

- The flatten primitive removes the outer layer of brackets from a vector of vectors:
  - flatten([[[0,1],[2,3]],[[4,5],[6,7]]])
    -> [[0, 1], [2, 3], [4, 5], [6, 7]]
  - flatten(flatten([[[0,1],[2,3]],[[4,5],[6,7]]]))
    -> [0, 1, 2, 3, 4, 5, 6, 7]
- flatten is assumed to take constant time on a VRAM.

Work/Depth for Primitives

Operation	Description	Work	Depth
dist(a,l)	*Create a sequence of* **a**s *of length* **l**.	1	1
#a	*Return length of sequence* **a**.	1	1
a[i]	*Return element at position* **i** *of* **a**.	1	1
[s:e]	*Return integer sequence from* **s** *to* **e**.	(e − s)	1
[s:e:d]	*Return integer sequence from* **s** *to* **e** *by* **d**.	(e − s)/d	1
sum(a)	*Return sum of sequence* **a**.	L(a)	log L(a)
write(d,a)	*Place elements* **a** *in* **d**.	L(a)	1
a ++ b	*Append sequences* **a** *and* **b**.	L(a) + L(b)	1
drop(a,n)	*Drop first* **n** *elements of sequence* **a**.	L(result)	1
interleave(a,b)	*Interleave elements of sequences* **a** *and* **b**.	L(result)	1
flatten(a)	*Flatten nested sequence* **a**.	L(result)	1

L(v) = length of vector v

Basic NESL Philosophy

- Try to convert algorithms to exploit optimized scan primitives as much as possible:
  - length of a Vector
  - sum of a Vector
  - permute(Vector, Index Vector)
  - p+(Vector1, Vector2) (pair-wise sum)
  - +-scan(Vector)
  - max-scan(Vector)
  - etc.

NESL Set-Patterns
(after SASL, KRC, Miranda)

- {*pattern* : *var* in *Vector*}
- {*pattern* : *var1* in *Vector1; var2* in *Vector2*}

- Example:
  - {f(x) : x in V} is essentially a map operation
  - {a + b : a in [1, 3]; b in [5, 9]} ==> [6, 12]

- Look like set operations.

- Run **in parallel**.

Divide&Conquer Implementation of Scans

```
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});
```

[Assumes associative op.]

O(n) work and O(log n) depth

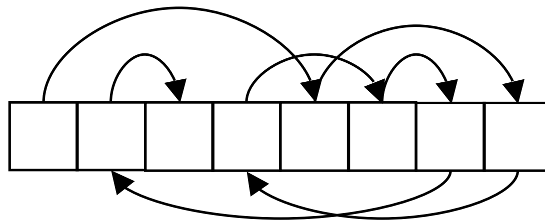
source for this and others: http://www.cs.cmu.edu/~scandal/nesl/alg-sequence.html

Implementation of List Ranking

- Wyllie's **Pointer Jumping** Technique
- As with prefix sum, this has many uses.
- Basic idea: in a chain of pointers stored in the common memory, the extremities of a chain can be determined in a way that **doubles the span** of a pointer at each step:
  - If a location points "N hops away" now, it will point "2N hops away" on the next step.
  - This is because concatenating two N-hop chains gives a 2N-hop chain.

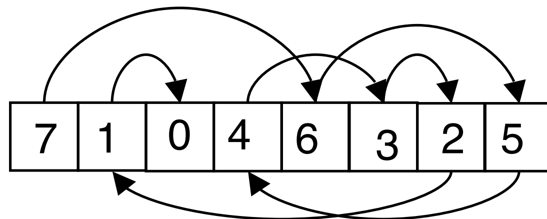
List-Ranking Problem

- A pointer-jumping application
- Given a chain of pointers, determine the rank of each element in the chain



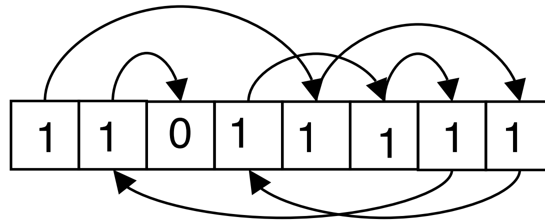
List-Ranking Problem

- A pointer-jumping application
- Given a chain of pointers, determine the rank of each element in the chain



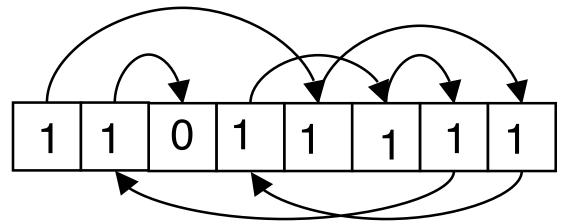
List-Ranking Step-by-Step

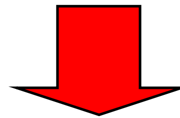
Step 0: If you point to something, your value is 1; else 0.

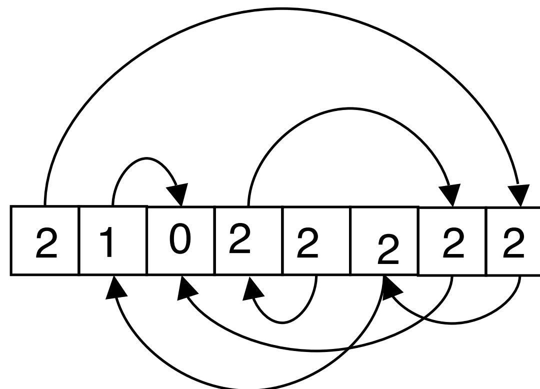


List-Ranking Step-by-Step

Step 1: If you point to something, add its value to yours.
Replace your pointer with its pointer.

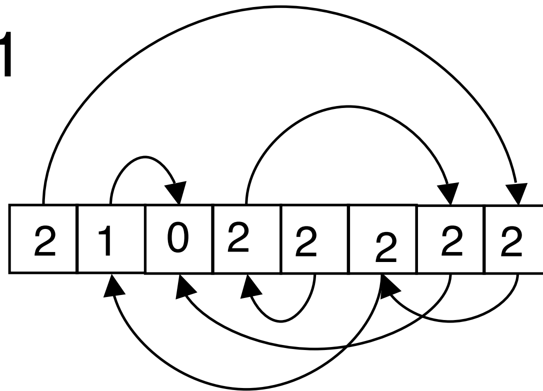


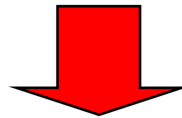


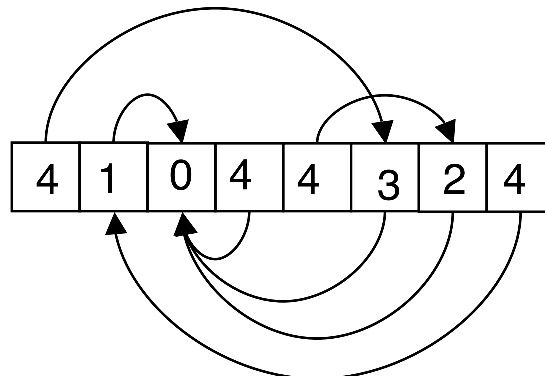


List-Ranking Step-by-Step

Step 2: repeat Step 1

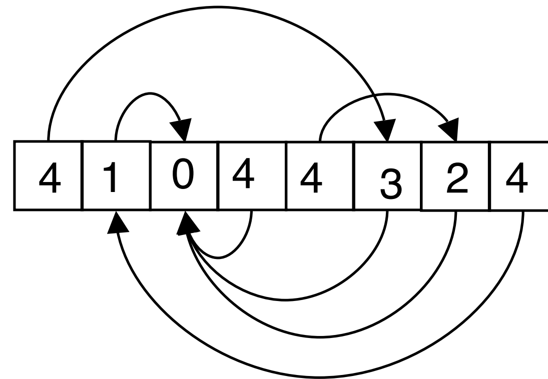


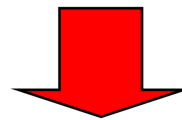


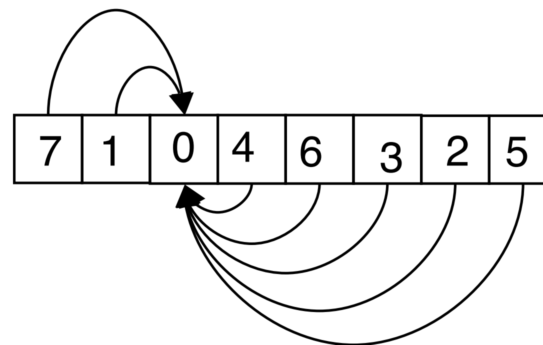


List-Ranking Step-by-Step

Step 3







Summary

- A list can be ranked in O(log n) time on with O(n log n) work.

NESL code for Pointer jumping

```
function pointer_jump(pt,val) =
let
   npt = pt->pt;
   nval = {v + nv: v in val; nv in val->pt};
in
   if eql(npt,pt) then val
   else pointer_jump(npt,nval);

function wyllie_list_rank(pt) =
let
   % set value to 1 everywhere except at the tails %
   val = {if i==pt then 0 else 1: pt; i in [0:#pt]};
in pointer_jump(pt,val);

% This example contains the two lists
     0 -> 2 -> 1 -> 5 -> 6
     4 -> 7 -> 3   %
wyllie_list_rank([2, 5, 1, 3, 7, 6, 6, 3]);
```

Parenthesis Matching (uses Ranking)

```
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);

parentheses_match("()(()())((()))");
```

Counting Sort

- Find rank of element by counting number less than.
- Work $O(n^2)$, depth $O(\log n)$ [Blelloch says depth $O(1)$, but can this be, with count requiring $O(\log n)$?]

```
function pair_comp((a1,a2),(b1,b2)) =
  if (a1 == b1) then a2 < b2 else a1 < b1;

function my_position(x,a) =
  count({pair_comp(y,x): y in a});

function counting_sort(a) =
let ai = {(a, i): a; i in [0:#a]}
in permute(a, {my_position(x,ai) : x in ai});

counting_sort([8, 14, -8, -9, 5, -9, -3, 0, 17, 19]);
```

Odd-Even Merge (not sort)

```
function odd_even_merge(a,b) =
if (#a > 1)
then
   let b = {odd_even_merge(a,b)
                 : a in [odd_elts(a),even_elts(a)]
                 ; b in [odd_elts(b),even_elts(b)]};
       odd, even = b[0],b[1];
   in take(even,1) ++
      flatten({[min(o,e),max(o,e)]
                   : o in drop(odd,-1)
               ; e in drop(even,1)}) ++
      take(odd,-1)
else
   if (a[0] < b[0]) then a++b else b++a;

odd_even_merge([2,8,9,11,12,18,22,22],
                     [1,5,6,11,13,14,19,21]);
```

O(log n) depth, with O(n log n) work (vs. O(n) work sequentially).

Bitonic Sort (power of 2 elements)

```
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]});

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]));

batcher_sort([8, 14, -8, -9, 5, -9, 17, 19]);
```

Work $O(n \log^2 n)$ and depth $O(\log^2 n)$.

FFT (Fast-Fourier Transform)

```
function fft(a,w,add,mult) =
if #a == 1 then a
else
  let r = {fft(b, even_elts(w), add, mult):
           b in [even_elts(a),odd_elts(a)]}
  in {add(a, mult(b, w)):
      a in r[0] ++ r[0];
      b in r[1] ++ r[1];
      w in w};

function complex_fft(a) =
let
    c = 2.*pi/float(#a);
    w = {cos(c*float(i)),sin(c*float(i)) : i in [0:#a]};
    add = ((ar,ai),(br,bi)) => (ar+br,ai+bi);
    mult = ((ar,ai),(br,bi)) => (ar*br-ai*bi,ar*bi+ai*br);
in fft(a,w,add,mult);

complex_fft([(2.,0.),(-1.,1.),(0.,0.),(-1.,-1.)]);
```

O(n log n) work, O(log n) depth

matrix-multiply

- matrix-multiply(A, B) =

```
        {
          { sum( {x*y: x in rowA; y in colB} )

                    : colB in transpose(B)
           }
          : rowA in A
        }
```

Sparse Matrix Representation

- Possible representation in NESL: Vector with column indices

$$A = \begin{matrix} 2.0 & -1.0 & 0 & 0 \\ -1.0 & 2.0 & -1.0 & 0 \\ 0 & -1.0 & 2.0 & -1.0 \\ 0 & 0 & -1.0 & 2.0 \end{matrix}$$

```
A = [[(0, 2.0), (1, -1.0)],
     [(0, -1.0), (1, 2.0), (2, -1.0)],
     [(1, -1.0), (2, 2.0), (3, -1.0)],
     [(2, -1.0), (3, 2.0)]]
```

- Multiply a sparse matrix A with a dense vector x:

```
{sum({v * x[i] : (i,v) in row}) : row in A};
```

Matrix Inversion (without Pivoting)

```
function Gauss_Jordan(A,i) =
if (i == #A) then A
else
   let
               (irow,Ap) = head_rest(A);
               val = irow[i];
               irow = {v/val : v in irow};
               Ap = {let scale = jrow[i]
                    in {v - scale*x : x in irow; v in jrow}
                    : jrow in Ap}

in Gauss_Jordan(Ap++[irow],i+1) $
```

```
function matrix_inverse(A) =
  let
    n = #A;

    % Pad the matrix with the identity matrix (i.e. A ++ I) %
    Ap = {row ++ rep(dist(0.,n),1.0,i):
                  row in A; i in [0:n]};

    % Run Gauss-Jordan elimination on padded matrix %
    Ap = Gauss_Jordan(Ap,0);

  % Drop the identity matrix at the front %
  in {drop(row,n) : row in Ap} $

A = [[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [1.0, 1.0, 2.0]];
AI = matrix_inverse(A);
matrix_inverse(AI);
```

Quicksort in NESL

```
function qsort(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 = {qsort(v) : v in [lesser, greater]}

    in result[0] ++ equal ++ result[1]
    $
```

Low-Level Quicksort Implementation using Segmented Scan

- [3, 1, 2, 7, 6, 11, 5, 4, 9, 10, 12, 8]
  pivot = 3, parallel comparisons
  [=, <, <, >, >, >, >, >, >, >, >, >]
  3-way split & segment
  [1, 2, | 3, | 7, 6, 11, 5, 4, 9, 10, 12, 8]

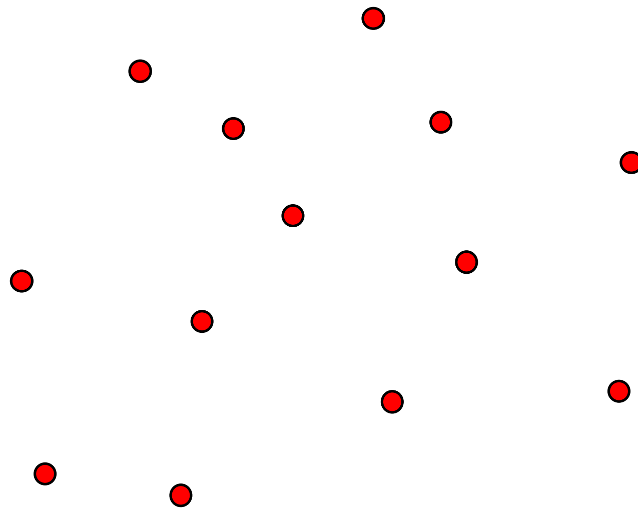
- segmented splits based on pivots
  [1, 2, | 3, | 6, 5, 4, 7, 11, 9, 10, 12, 8]
  [1, | 2, | 3, | 4, 5, 6, | 7, | 9, 10, 8, 11, 12]
- etc.

Quicksort analysis

- Worst case O(n) depth
- Average case O(log n) depth
- Average case work O(n log n)

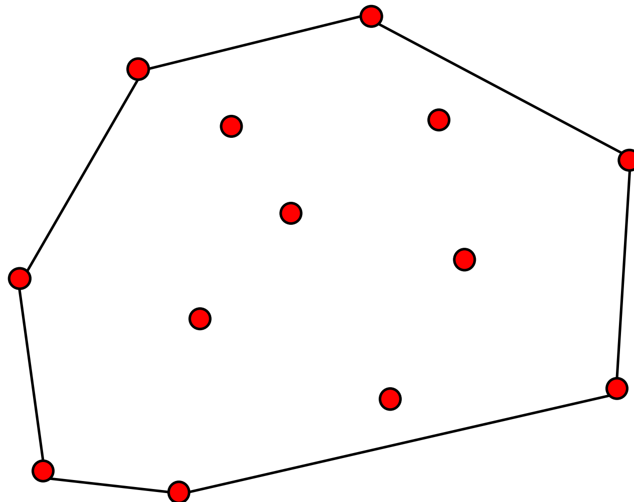
Convex Hull Algorithm ("Quickhull")

- Problem: Given n points in the plane, determine the subset that lie on the perimeter of the smallest convex region containing all of the points.



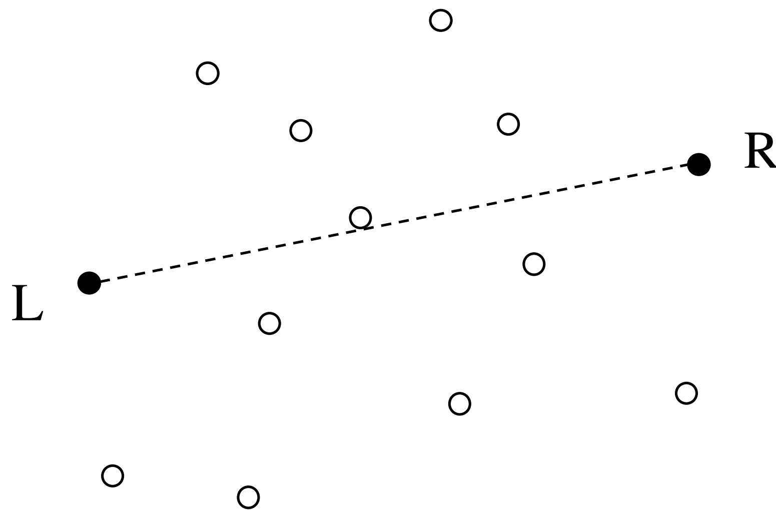
Convex Hull Algorithm ("Quickhull")

- Problem: Given n points in the plane, determine the subset that lie on the perimeter of the smallest convex region containing all of the points.



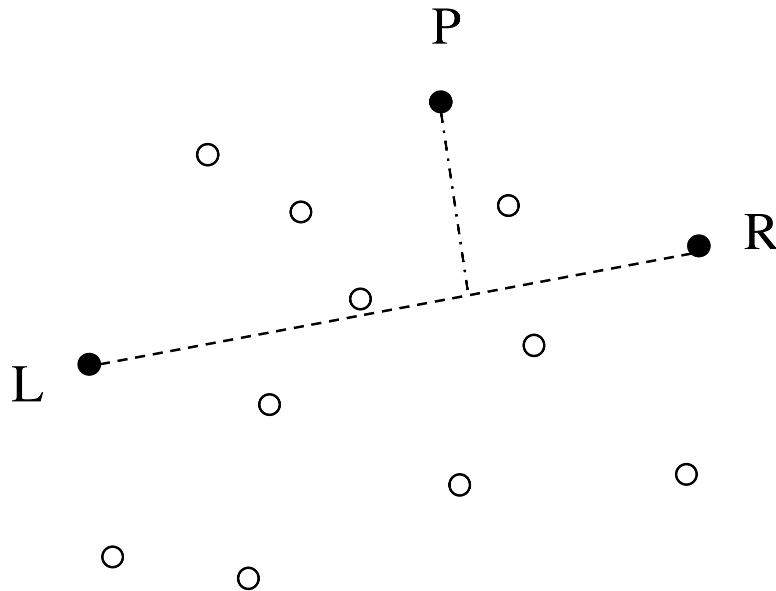
Convex Hull Algorithm ("Quickhull")

- Begin by finding the two extrema, L and R, in the x dimension.
- L and R will be in the convex hull.
- Imagine a line between these extrema.



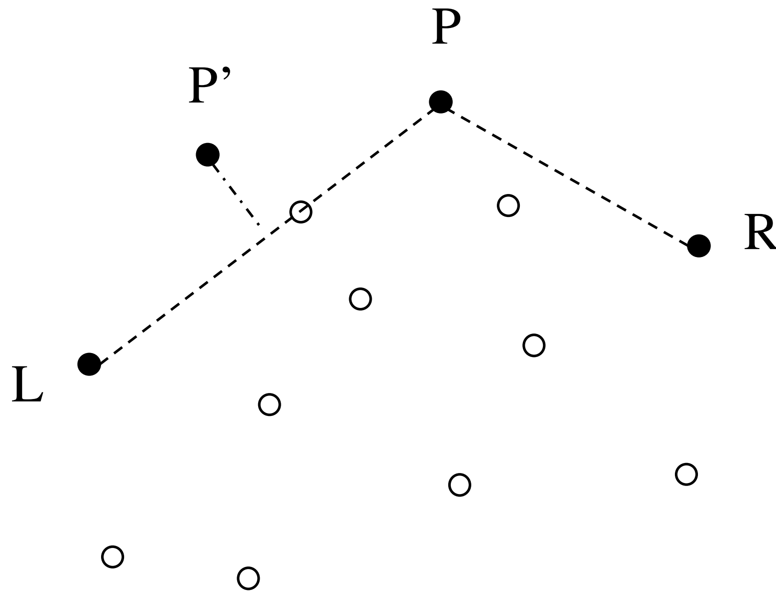
Convex Hull Algorithm ("Quickhull")

- Find the point P above and farthest from line LR, if any.
- P will also be in the convex hull.



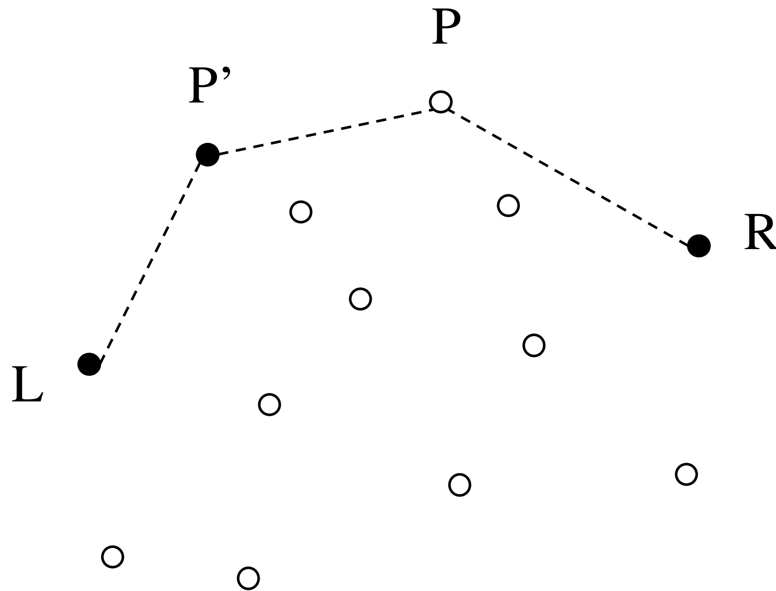
Convex Hull Algorithm (“Quickhull”)

- Repeat the process with lines LP and PR, until there is no point outside.
- The new points, P’, etc. are in the convex hull.



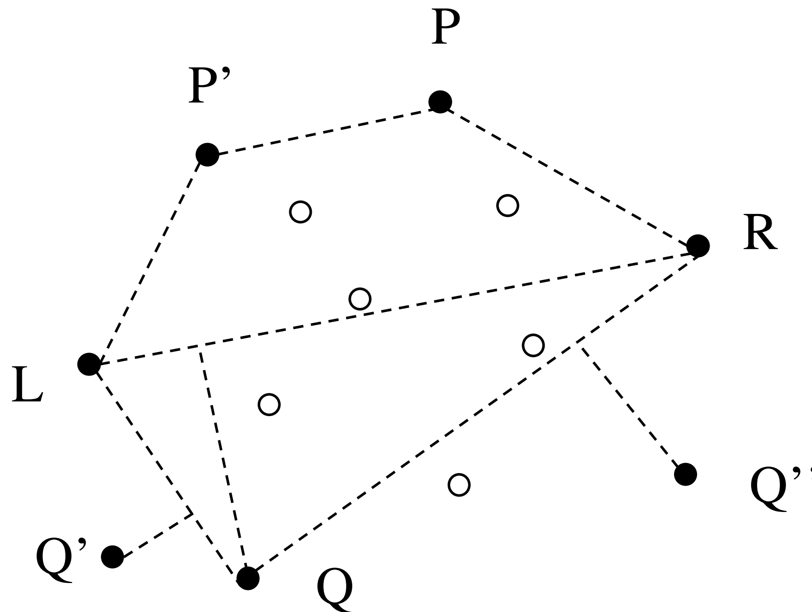
Convex Hull Algorithm ("Quickhull")

- Repeat the process with lines LP and PR, until there is no point outside.
- The new points, P', etc. are in the convex hull.

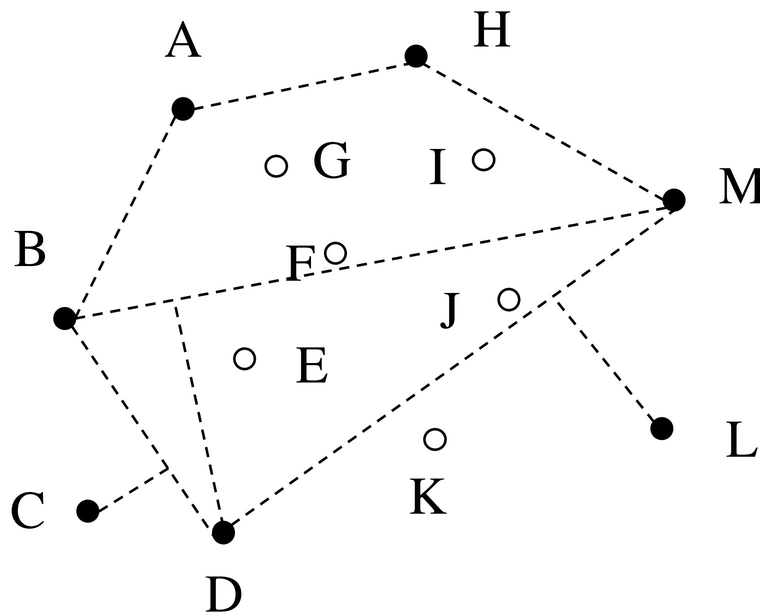


Convex Hull Algorithm ("Quickhull")

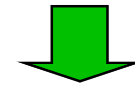
- **Meanwhile**, also be doing this with points on the other side of LR in parallel (call those points Q, Q', …)



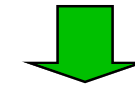
Representation as NESL Lists



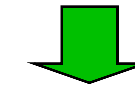
[A B C D E F G H I J K L M]



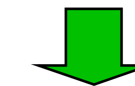
[B [A F G H I] [C D E J K L] M]



[B [A] H [C] D [E J K] L M]



[B A H C D [K] L M]



[B A H C D L M]

NESL Program for Quickhull (1)

```
% Used to find the distance of a point (o) from a line (line). %
function cross_product(o,line) =
let (xo,yo) = o;
    ((x1,y1),(x2,y2)) = line;
in (x1-xo)*(y2-yo) - (y1-yo)*(x2-xo);

% Given two points on the convex hull (p1 and p2), hsplit finds
  all the points on the hull between p1 and p2 (clockwise),
  inclusive of p1 but not of p2. %

function hsplit(points,(p1,p2)) =
let cross = {cross_product(p,(p1,p2)): p in points};
    packed = {p in points; c in cross | plusp(c)};           % plusp(c) means c > 0 %
in if (#packed < 2) then [p1] ++ packed
   else
     let pm = points[max_index(cross)];
     in flatten({hsplit(packed,ends): ends in [(p1,pm),(pm,p2)]});
```

NESL Program for Quickhull (2)

```
% Finds the points with minimum and maximum x coordinates, and then
  finds the upper and lower convex hull: the part clockwise from minx to
  maxx (upper) and clockwise from maxx to minx (lower). %

function convex_hull(points) =

let x = {x : (x,y) in points};
    minx = points[min_index(x)];
    maxx = points[max_index(x)];

in hsplit(points,minx,maxx) ++ hsplit(points,maxx,minx);
```

Quickhull Output

points
= [(3.0, 3.0), (2.0, 7.0), (0.0, 0.0), (8.0, 5.0), (4.0, 6.0),
(5.0, 3.0), (9.0, 6.0), (10.0, 0.0)] : [(float, float)]

result
[(0.0, 0.0), (2.0, 7.0), (9.0, 6.0), (10.0, 0.0)] : [(float, float)]

Analysis

- Similar to quicksort
- For "well-distributed" set of points, requires O(log n) depth.
- In worst case, can require O(n) depth.

Prime Generation

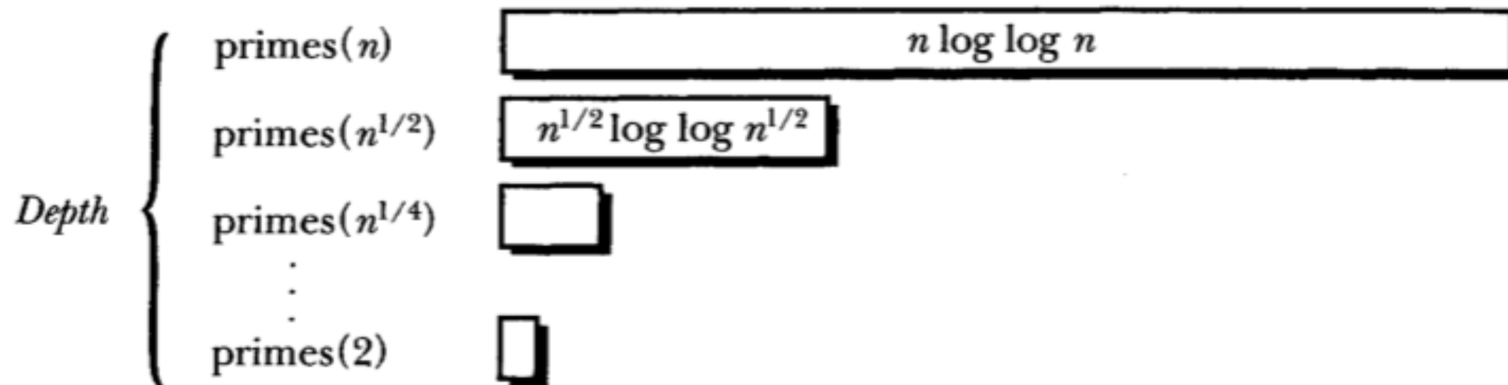
- Idea (Blelloch, CACM, March 1996):

  To generate primes up to n, first generate primes to sqrt(n), then use **sieving**.

- Apply the idea recursively.

- Yields O(n log log n) work, O(log log n) depth.

Prime Generation



Prime Generation

```
function primes(n) =
if n == 2 then ([] int)
else
      let sqr_primes = primes(isqrt(n));
          composites = {[2*p:n:p]: p in sqr_primes};
          flat_comps = flatten(composites);
          flags          = write(dist(true, n), {(i,false): i in flat_comps});
          indices        = {i in [0:n]; fl in flags | fl}
     in drop(indices, 2);
```

Example for primes(20):

```
sqr_primes  =  [2,3]
composites  =  [[4,6,8,10,12,14,16,18] , [6,9,12,15,18]]
flat_comps  =  [4,6,8,10,12,14,16,18,6,9,12,15,18]
flags       =  [t,t,t,t,f,t,f,t,f,f,f,t,f,t,f,f,f,t,f,t]
indices     =  [0,1,2,3,5,7,11,13,17,19]
result      =  [2,3,5,7,11,13,17,19]
```

3D Barnes-Hut

- See
  http://www.cs.cmu.edu/~scandal/nesl/algorithms/Barnes_Hut.nesl
  (about 360 lines)

NESL Reference Card (1)

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

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

NESL Reference Card (2)

[and there's more]

Basic Sequence Functions	O(1) depth
Basic Operations	
`#a`	Length of a
`a[i]`	i^{th} element of a
`dist(a,n)`	Create sequence of length n with a in each element.
`zip(a,b)`	Elementwise zip two sequences together into a sequence of pairs.
`[s:e]`	Create sequence of integers from s to e (not inclusive of e)
`[s:e:d]`	Same as `[s:e]` but with a stride d.
Scans	
`plus_scan(a)`	Execute a scan on a using the + operator
`min_scan(a)`	Execute a scan on a using the `minimum` operator
`max_scan(a)`	Execute a scan on a using the `maximum` operator
`or_scan(a)`	Execute a scan on a using the or operator
`and_scan(a)`	Execute a scan on a using the and operator
Reductions	
`sum(a)`	Sum the elements of a
`max_val(a)`	Return maximum value of a
`min_val(a)`	Return minimum value of a
`any(a)`	Return true if any values of a are true.
`all(a)`	Return true only if all values of a are true.
`count(a)`	Count number of true values in a.
`max_index(a)`	Return position (index) of maximum value.
`min_index(a)`	Return position (index) of minimum value.
Reordering Functions	
`read(a, i)`	Read values in a from indices i
`write(a, iv_pairs)`	Write values in a using integer values pairs in `iv_pairs`
`permute(a,i)`	Permute elements in a to indices in i
`rotate(a,i)`	Rotate sequence a by i locations
`reverse(a)`	Reverse order of sequence a.
`drop(a,i)`	Drop first i elements of a.
`take(a,i)`	Take first i elements of a.
`odd_elts(a)`	Return the odd elements of a
`even_elts(a)`	Return the even elements of a
`interleave(a,b)`	Interleave the elements of a and b.
`subseq(a,i,j)`	Return the subsequence of a from position i to j (not inclusive of j)
`a -> i`	Same as `read(a, i)`

NESL Lessons Learned

- From slides, 2006, by Blelloch
- Declarative Programming Languages for Multicore Architectures 2006
- http://glew.org/damp2006/Nesl.ppt

Lesson 1: Sequential Semantics

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

Lesson 2: Cost Semantics

- Need a way to analyze cost, at least approximately, without knowing details of the implementation.

- Any cost model based on processors is not going to be portable – too many different kinds of parallelism.

Lesson 3: Too Much Parallelism

Needed ways to back out of parallelism

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

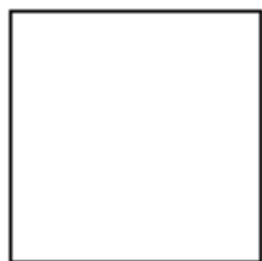
ZPL

- Conceived by Larry Snyder at Univ. of Washington.
- The last word in data parallel languages? Zettaflops?
- Oriented toward grid computations.
- Adds concept of “regions” to language.

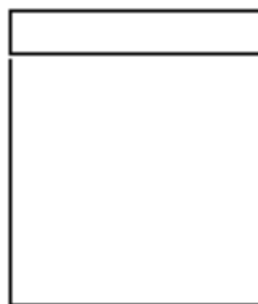
ZPL Stencil Code with Region

```
program Jacobi;  /* Nearest neighbor by L. Snyder, 1994 */
  config var n : integer = 512; -- Declarations
  delta : float = 0.000001;

  region R = [1..n, 1..n];
  var A, Temp: [R] float;
  err : float;

  direction north = [-1, 0];
            east  = [ 0, 1];
            west  = [ 0,-1];
            south = [ 1, 0];

procedure Jacobi();
  begin
    [R] A := 0.0; -- Initialization
    [north of R] A := 0.0;
    [east of R]  A := 0.0;
    [west of R]  A := 0.0;
    [south of R] A := 1.0;

    [R] repeat -- Body
      Temp := (A@north + A@east + A@west + A@south)/4.0;
      err := max<< abs(A - Temp);
      A := Temp;
     until err < delta;
    end;
```

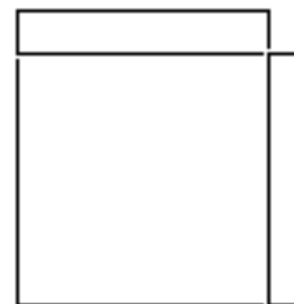
Initialization, Broadcasting



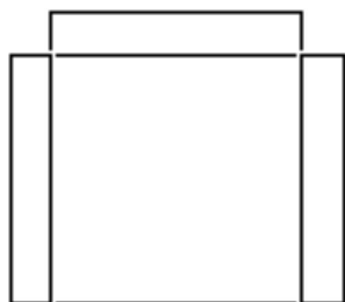
A



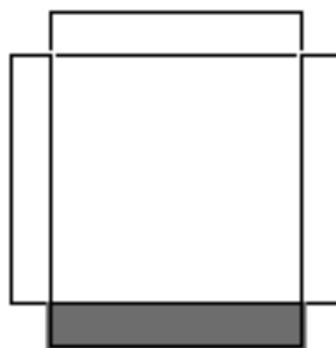
[north of R] A:=0;



[east of R] A:=0;



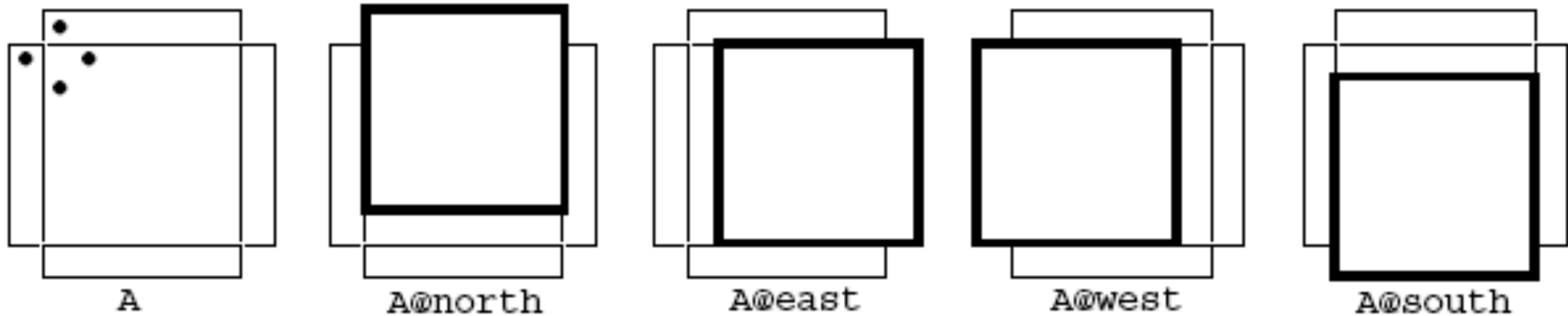
[west of R] A:=0;



[south of R] A:=1;

Updating

```
Temp := (A@north + A@east
          + A@west + A@south)/4.0;
A := Temp;
```



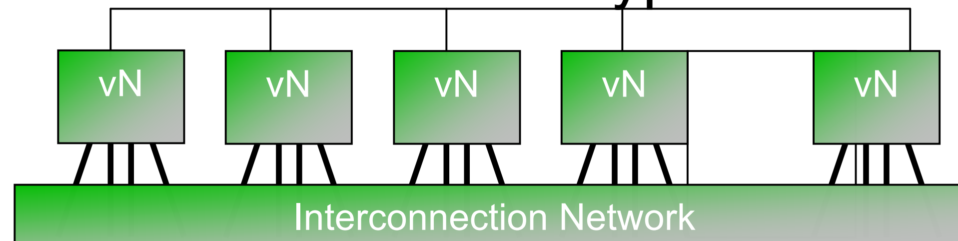
Region Stencil Computation

```
NN := TW@^nw + TW@^no + TW@^ne
    + TW@^w  +          TW@^e
    + TW@^sw + TW@^so + TW@^se;

TW := (TW & NN = 2) | (NN = 3);
```

```
[1..m,1..p] for k := 1 to n do
      C += (>>[ ,k] A)*(>>[k, ]B); end;
```

ZPL Is Built On The “CTA”

- ZPL uses the “Candidate Type Architecture” (Snyder)



  - Invented long ago, 100% contemporary
    - *P* processors
    - Memory is local (latency $\lambda = 1$) or non-local ($\lambda >> 1$)
  - Many contemporary parallel machines implement CTA => portability and performance

Data-Parallel Haskell

- A group headed by Manuel M. T. Chakravarty from Univ. of New South Wales and Simon Peyton-Jones from Microsoft Research are working on Haskell or a variation as a data-parallel language.
- Nepal -- Nested Data-Parallelism in Haskell
- One of the thrusts of this work is **program transformation** for parallel execution.
- Inspiration from NESL is acknowledged.

Haskell

- Ultra-pure functional language
- Strongly-typed
- Type inference
- Lazily-evaluated
- List-comprehension notation available
- GHC = Glasgow Haskell Compiler
- (Concurrent Haskell includes explicit parallelism)

List and Array Comprehensions

- List comprehension

  [F x | x <- L]   suggests $\{F(x) \mid x \in L\}$

- Array comprehension:

  [: F x | x <- L :]

Note: All Haskell functions are 1-ary;
Currying is used for the general case.

Type Signatures of Some Array Operations in Haskell

```
(!:) :: [:a:] -> Int -> a
sliceP :: [:a:] -> (Int,Int) -> [:a:]
replicateP :: Int -> a -> [:a:]
mapP :: (a->b) -> [:a:] -> [:b:]
zipP :: [:a:] -> [:b:] -> [:(a,b):]
zipWithP :: (a->b->c) -> [:a:] -> [:b:] -> [:c:]
filterP :: (a->Bool) -> [:a:] -> [:a:]
concatP :: [:[:a:]:] -> [:a:]
concatMapP :: (a -> [:b:]) -> [:a:] -> [:b:]
unconcatP :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP :: [:[:a:]:] -> [:[:a:]:]
expandP :: [:[:a:]:] -> [:b:] -> [:b:]
combineP :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
splitP :: [:Bool:] -> [:a:] -> ([:a:], [:a:])
```

Source:
http://www.cse.unsw.edu.au/~chak/papers/PLKC08.html
Harnessing the Multicores: Nested Data Parallelism in Haskell
Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, Manuel M. T. Chakravarty
Foundations of Software Technology and Theoretical Computer Science (Bangalore) 2008.
Editors: R. Hariharan, M. Mukund, V. Vinay

Also gives some details of transformations described in the following slides.

Example: Primes (as shown previously in NESL)

```
primesUpTo :: Int -> [: Int :]
primesUpTo 1 = [: :]
primesUpTo 2 = [: 2 :]
primesUpTo n = smallers +:+
                    [: x | x <- [: ns+1..n :]
                     , not (anyP (‘divides‘ x) smallers) :]
where
ns = intSqrt n
smallers = primesUpTo ns
```

Source:
Simon Peyton Jones and Satnam Singh
A Tutorial on Parallel and Concurrent Programming in Haskell
Lecture Notes from Advanced Functional Programming
Summer School 2008

Shows how some NESL ideas can be adapted to Haskell

- The following (blue background) slides are excerpts from

  Data Parallel Haskell

  by SP Jones, R Leshchinskiy, G Keller, M Chakravarty

  http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

Motivation

Multicore

Parallel programming essential

Task parallelism

- Explicit threads
- Synchronise via locks, messages, or STM

Modest parallelism
Hard to program

Data parallelism

Operate simultaneously on bulk data

Massive parallelism
Easy to program

- Single flow of control
- Implicit synchronisation

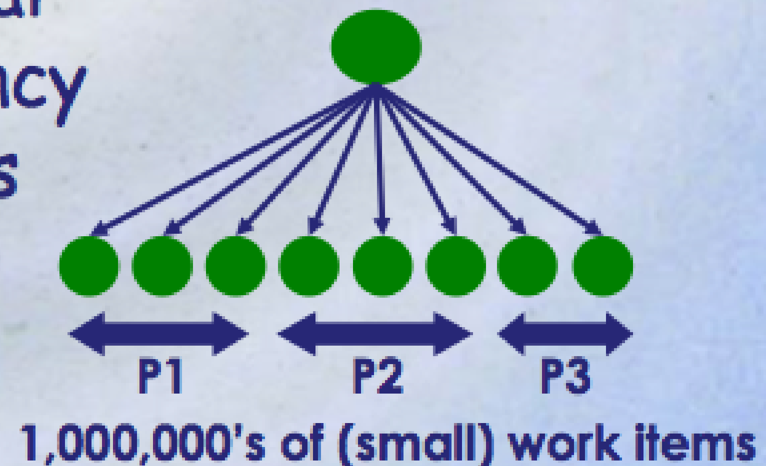
Flat data parallelism

- Apply sequential operation to bulk data
- Widely used, well understood, well supported

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- "something" is sequential
- Single point of concurrency
- Great for dense matrices

e.g. HPF, MPI, MapReduce, Matlab's Parallel Toolbox

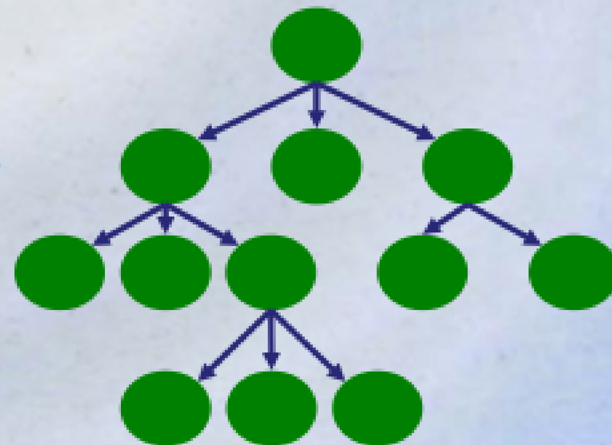


Nested data parallel

- Main idea: **allow "something" to be parallel**

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- Now the parallelism structure is recursive, and un-balanced
- Hard to implement!



Still 1,000,000's of (small) work items

Nested DP is great for **programmers**

- Fundamentally more modular
- Opens up a much wider range of applications:
  - Sparse arrays, variable grid adaptive methods (e.g. Barnes-Hut)
  - Divide and conquer algorithms (e.g. sort)
  - Graph algorithms (e.g. shortest path, spanning trees)
  - Physics engines for games, computational graphics (e.g. Delauny triangulation)
  - Machine learning, optimisation, constraint solving

Nested DP is tough for **compilers**

- ...because the concurrency tree is both irregular and fine-grained
- But it can be done! NESL (Blelloch 1995) is an existence proof
- Key idea: "flattening" transformation:
  nested DP ➡ flat DP

What does DPH add?

- Parallel arrays
  - *efficient unboxed representations for all Haskell data types*
- Parallelized library of array operations
- Convenient array comprehension syntax
- Source-to-source transformation pipeline
  - *Turns inefficient Nested Data parallelism into optimized Flat Data parallelism*
- "Gang Parallelism" execution model

Parallel array comprehensions

[:Float:] is the type of parallel arrays of Float

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

sumP :: [:Float:] -> Float

An array comprehension: "the array of all f1*f2 where f1 is drawn from v1 and f2 from v2"

Operations over parallel array are computed in parallel; that is the only way the programmer says "do parallel stuff"

NB: no locks!

Sparse vector multiplication

A sparse vector is represented as a vector of (index,value) pairs

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul sv v = sumP [: f*(v !: i) | (i,f) <- sv :]
```

v !: i gets the i'th element of v

Parallelism is proportional to length of sparse vector

Sparse matrix multiplication

A sparse matrix is a vector of sparse vectors

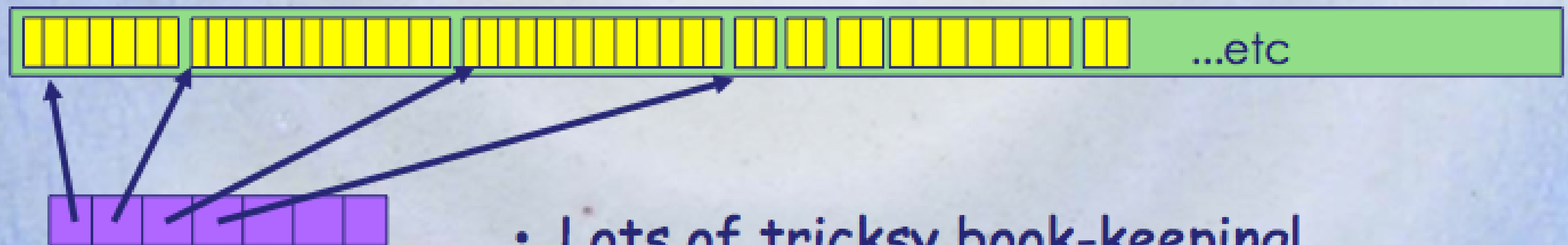
```
smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul row v | row <- sm :]
```

Nested data parallelism here!

We are calling a parallel operation, svMul, on every element of a parallel array, sm

The flattening transformation

- Concatenate sub-arrays into one big, flat array
- Operate in parallel on the big array
- Segment vector keeps track of where the sub-arrays are



- Lots of tricksy book-keeping!
- Possible to do by hand (and done in practice), but very hard to get right
- Blelloch's NESL showed it could be done systematically

Transformation Pipeline

1. **Desugaring**
   - comprehensions -> function calls
2. **Vectorization/Flattening**
   - maps Nested Data parallelism to Flat Data parallelism
3. **Distribution**
   - splits computation between a gang of threads
4. **Fusion**
   - eliminate intermediate arrays and unnecessary synchronization points in sequential code executed on each thread

Fusion

- Flattening is not enough

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

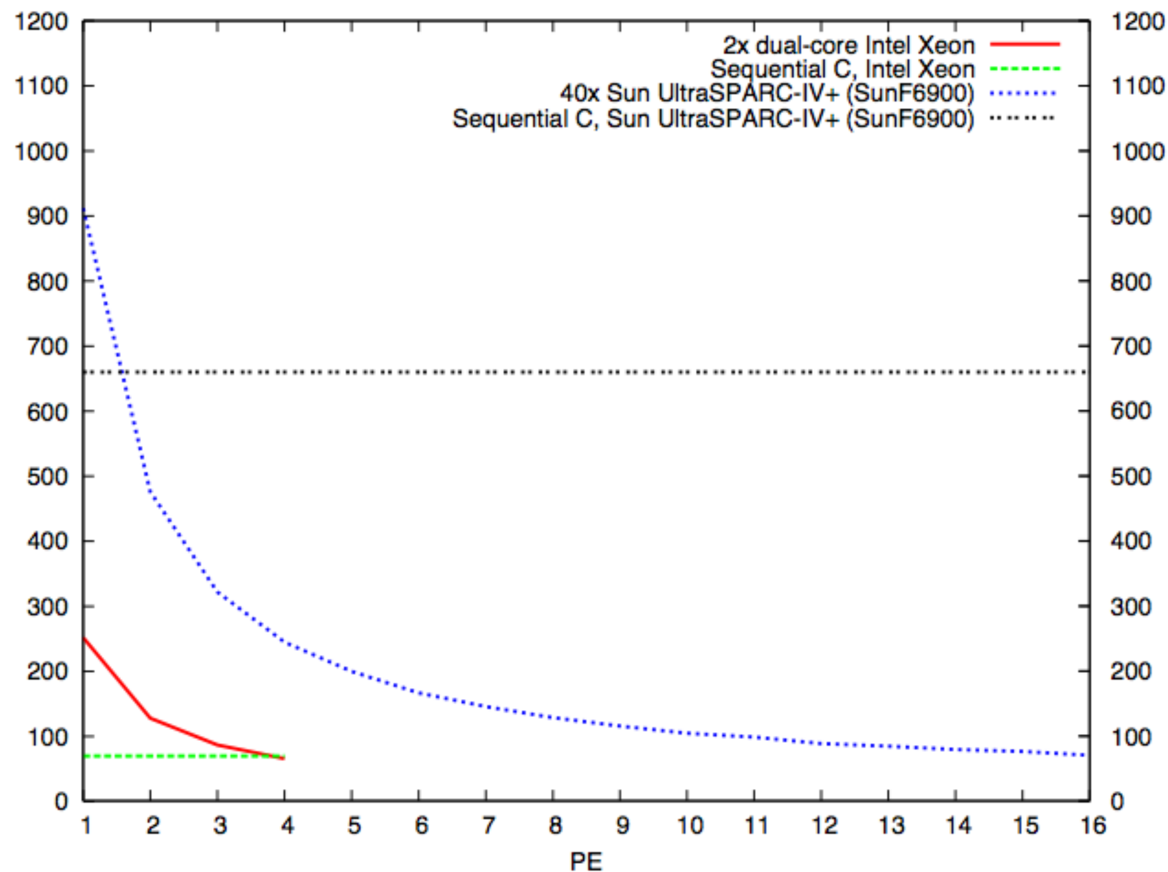
- Do not
  1. Generate [: f1*f2 | f1 <- v1 | f2 <- v2 :]
     (big intermediate vector)
  2. Add up the elements of this vector
- Instead: multiply and add in the same loop
- That is, **fuse** the multiply loop with the add loop
- Very general, aggressive fusion is required

Summary

- Data parallelism is the only way to harness 100's of cores
- Nested DP is great for programmers: far, far more flexible than flat DP
- Nested DP is tough to implement
- But we (think we) know how to do it
- Functional programming is a massive win in this space
- Prototype in current GHC release (6.10.1)

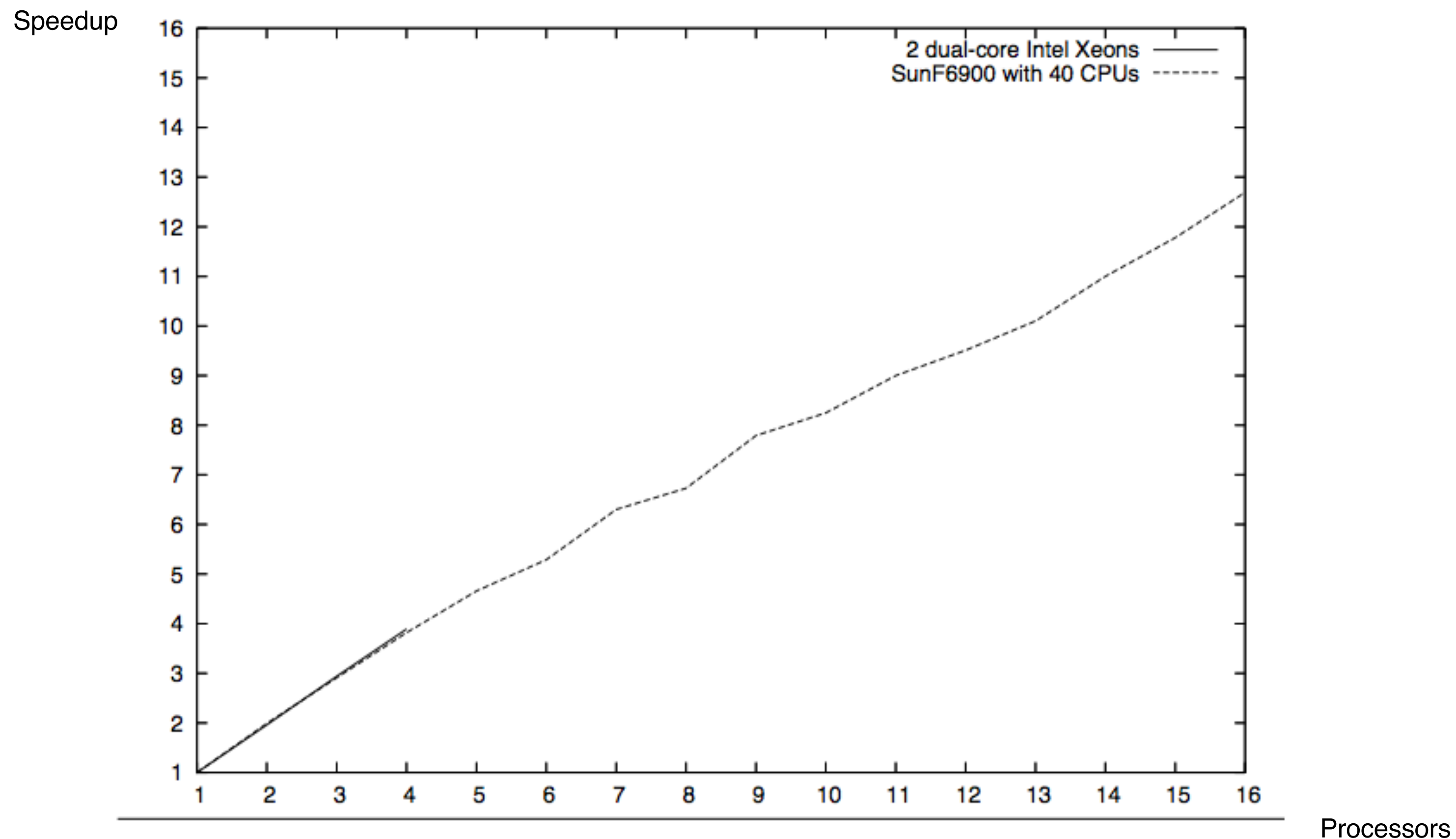
http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

Perfomance? sparse matrix-vector multiply



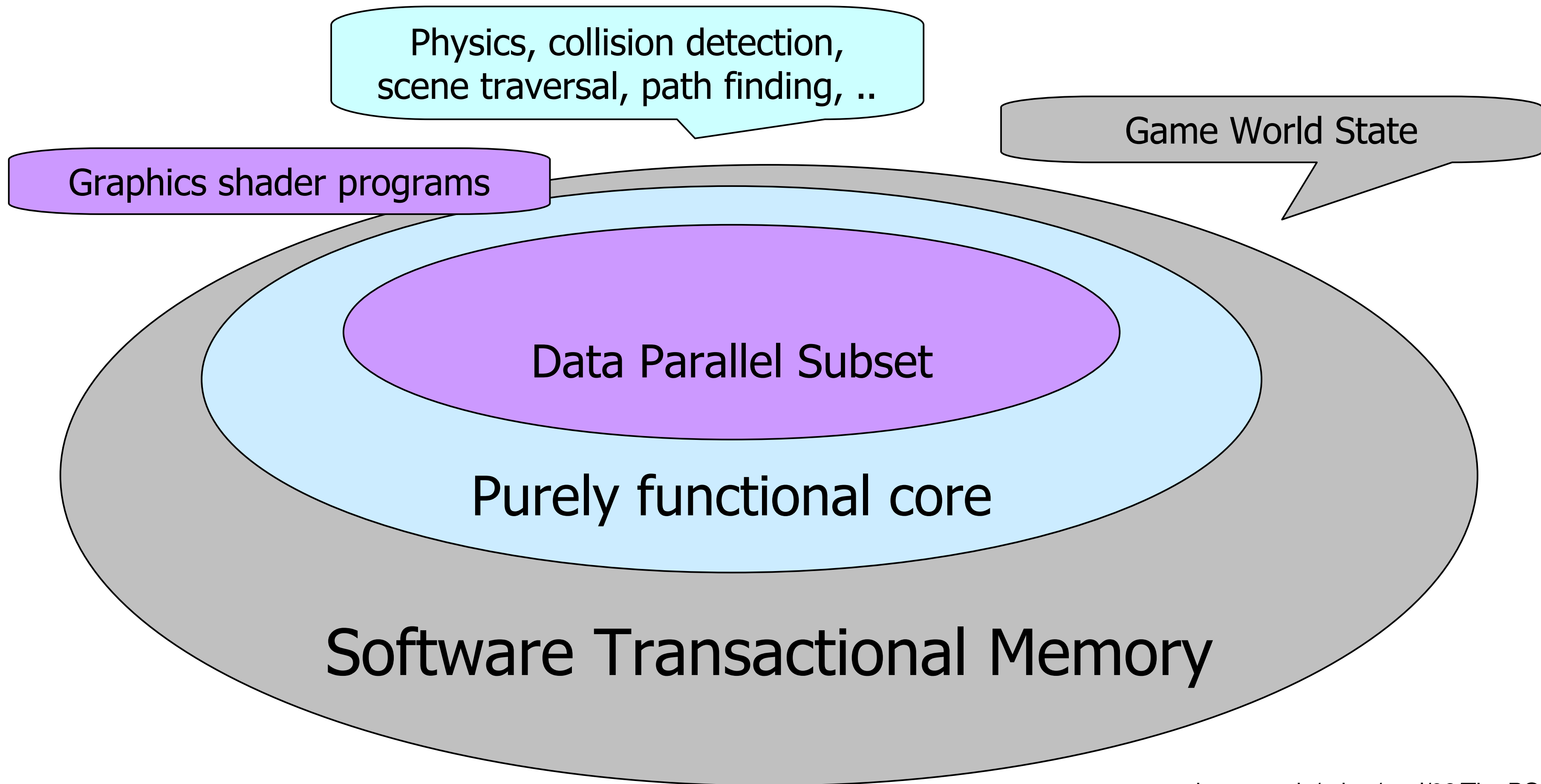
source: Chakravarty, Leshchinskiy, Jones, Keller, Marlow: Compiling Nested Data Parallelism by Program Transformation
http://dataparallel.googlegroups.com/web/UNSW CGO DP 2007.pdf

Speedup?
sparse matrix-vector multiply



source: Chakravarty, Leshchinskiy, Jones, Keller, Marlow: Data Parallel Haskell: a status report
www.cs.cmu.edu/~damp/finalPapers/chakravarty.pdf

Example: Data Parallelism in Game Programming



source: www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt
Tim Sweeney
The Next Mainstream Programming Language: A Game Developer's Perspective

Example: Data Parallelism in Game Programming

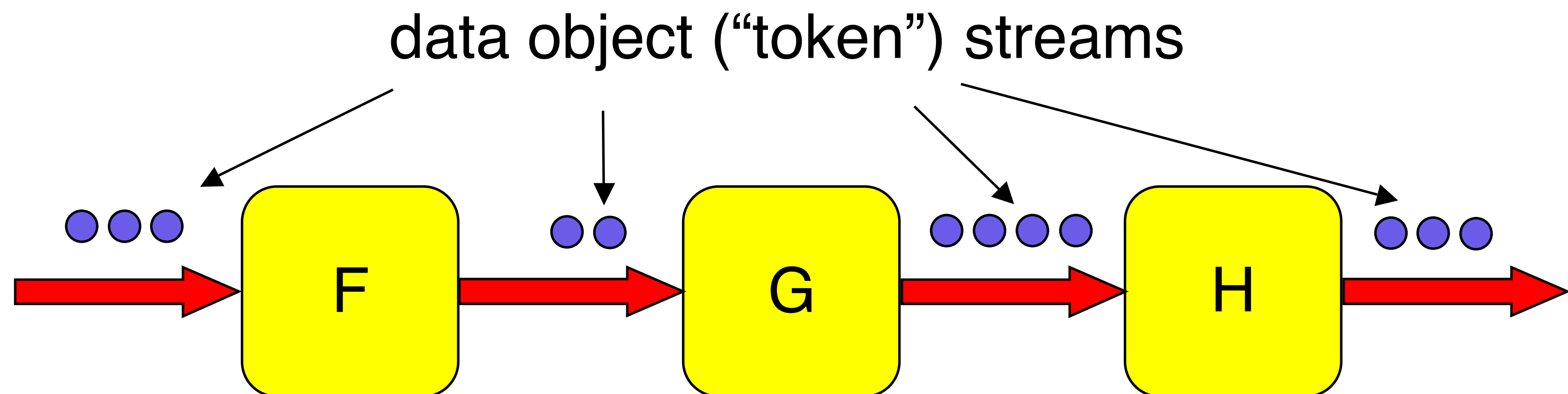
	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS
Parallelism	**Software Transactional Memory (STM)**	**Implicit Thread Parallelism**	**Implicit Data Parallelism**

source: www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt
Tim Sweeney
The Next Mainstream Programming Language:
A Game Developer's Perspective

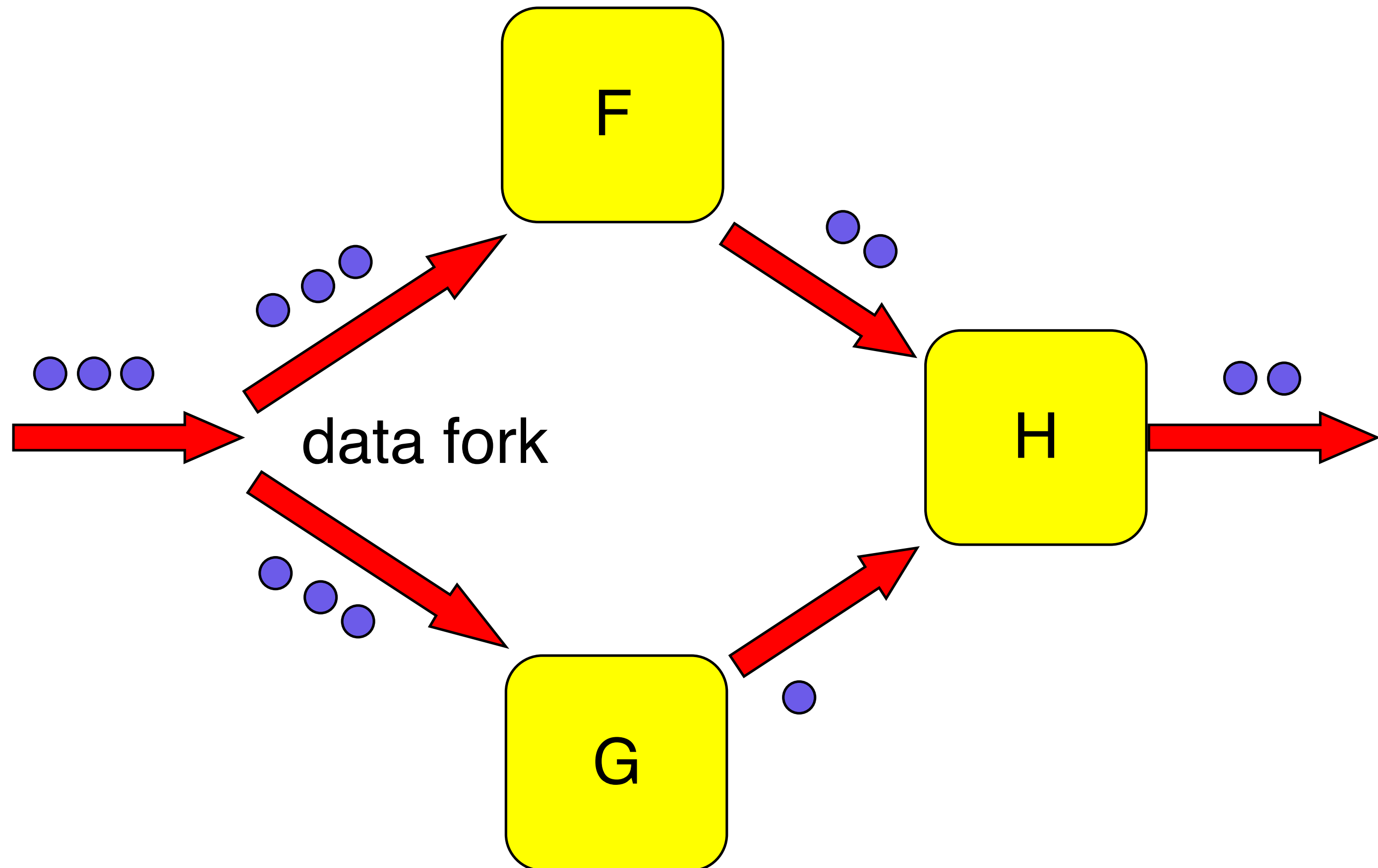
Dataflow Parallelism

- Category of functional programming
- No explicit control-flow
- Natural for pipelining, concurrency
- Two general viewpoints:
  - **Data-driven**: Data pushed through the system
  - **Demand-driven**: Results pulled from the system

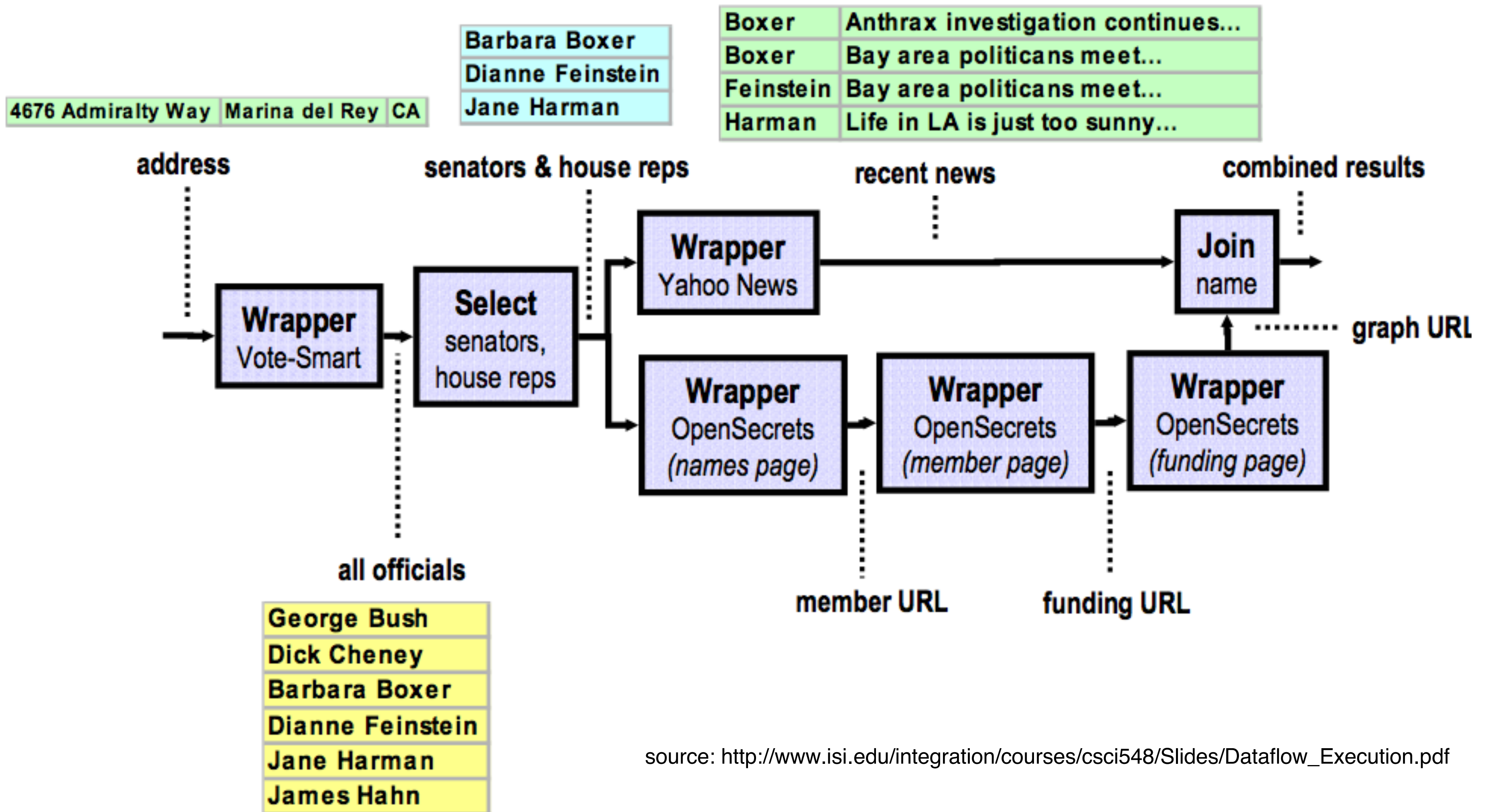
Pipelined Dataflow Graph



Concurrent Dataflow Graph

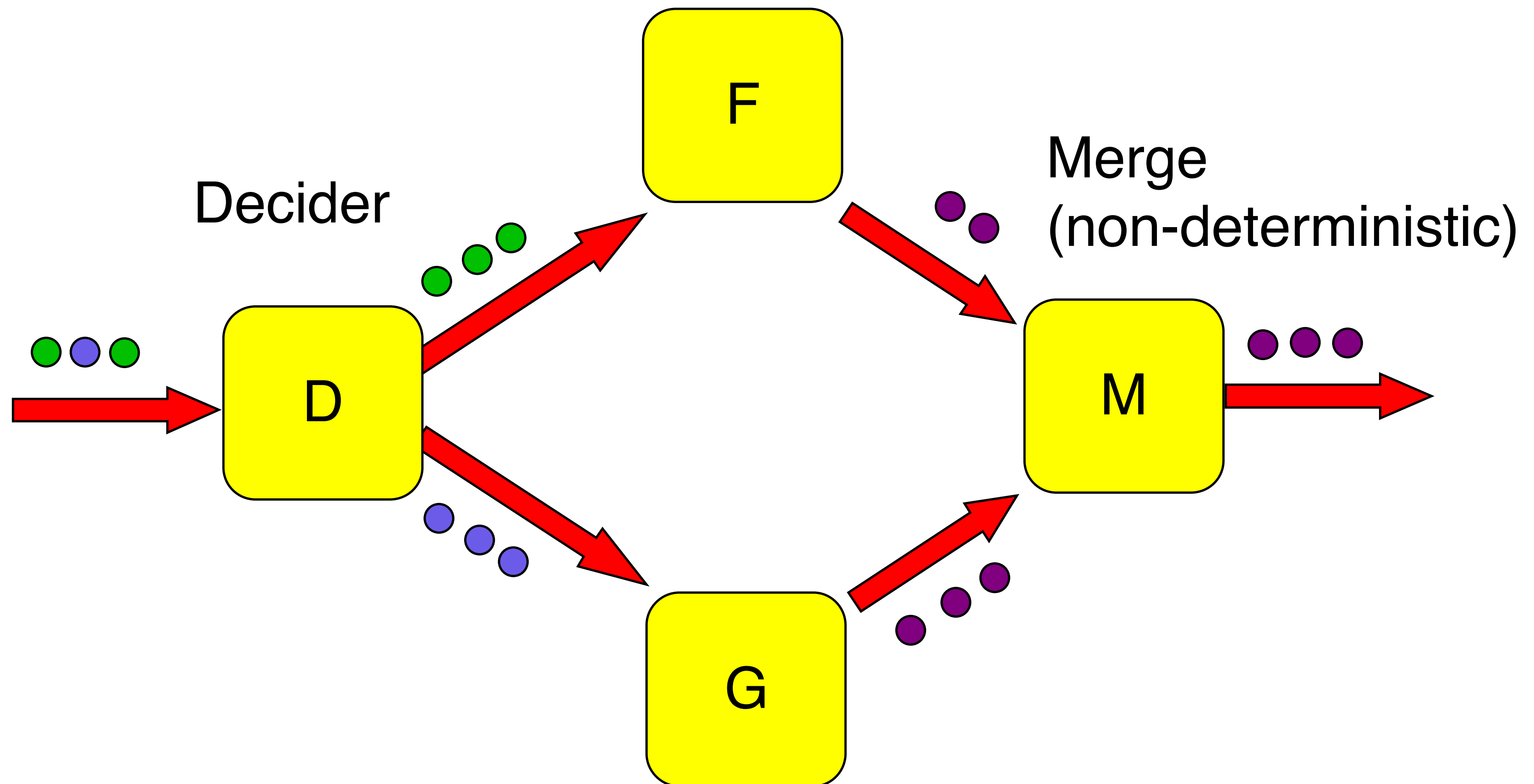


Dataflow Graph for an Info Tracking App



source: http://www.isi.edu/integration/courses/csci548/Slides/Dataflow_Execution.pdf

Dataflow Graph with Decision&Merge



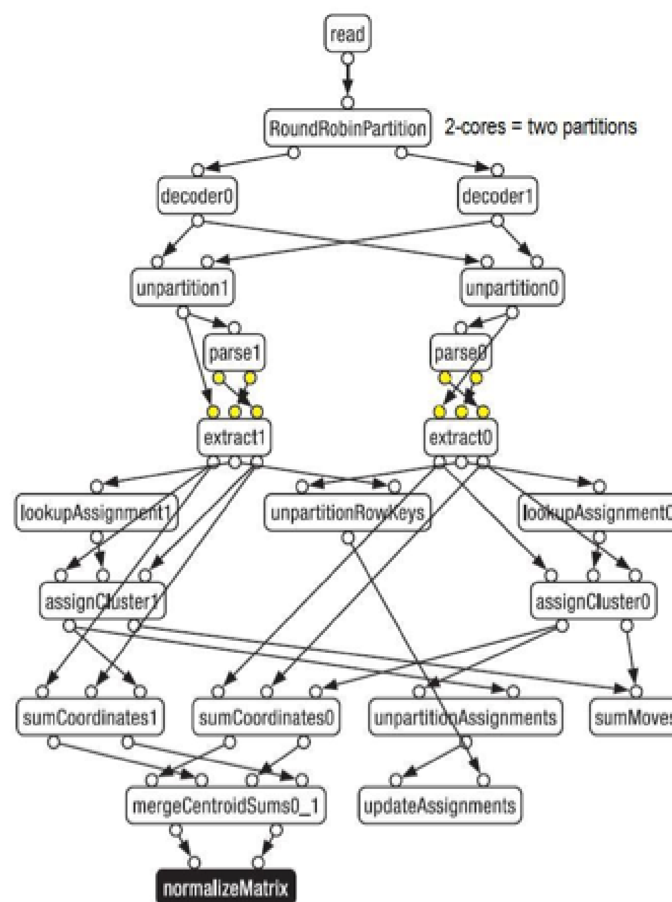
Once you have these, you can do Dataflow Loops.

Example from a Netflix Prize Entrant

Dataflow

1) Assign cluster: calc distances

2) Sum coordinates, merge centroids, normalize: calc centroids



Advantage of Dataflow

- Dataflow reduces programming to plumbing.
- Unfortunately, plumbers get paid more, and don’t take their work home with them.