CSE524 Parallel Computation

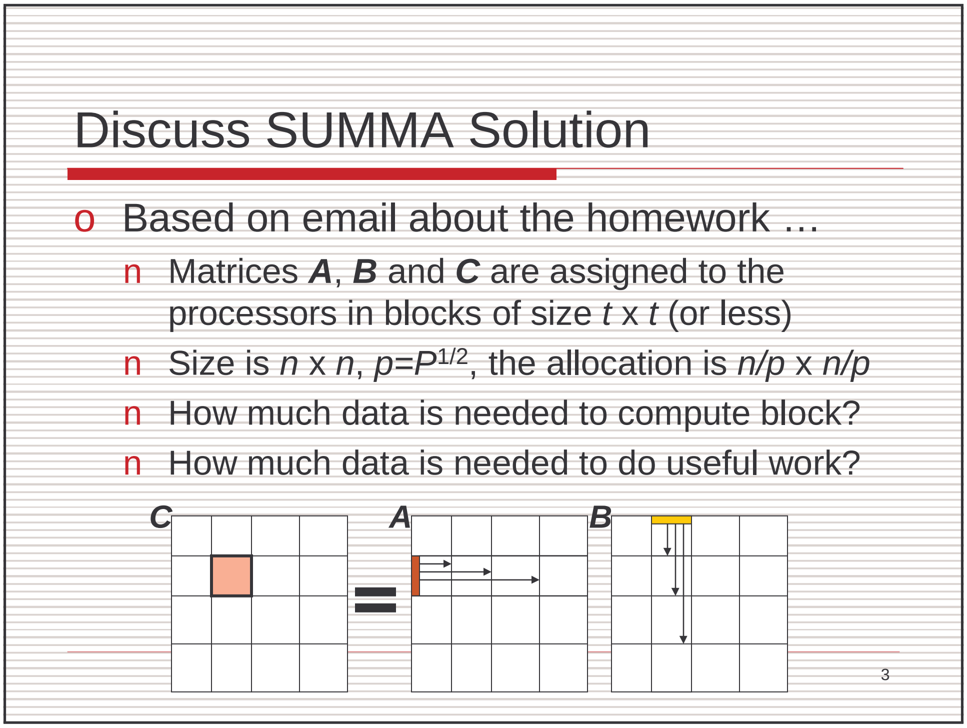
Lawrence Snyder
www.cs.washington.edu/CSEp524

1 May 2007

Announcements

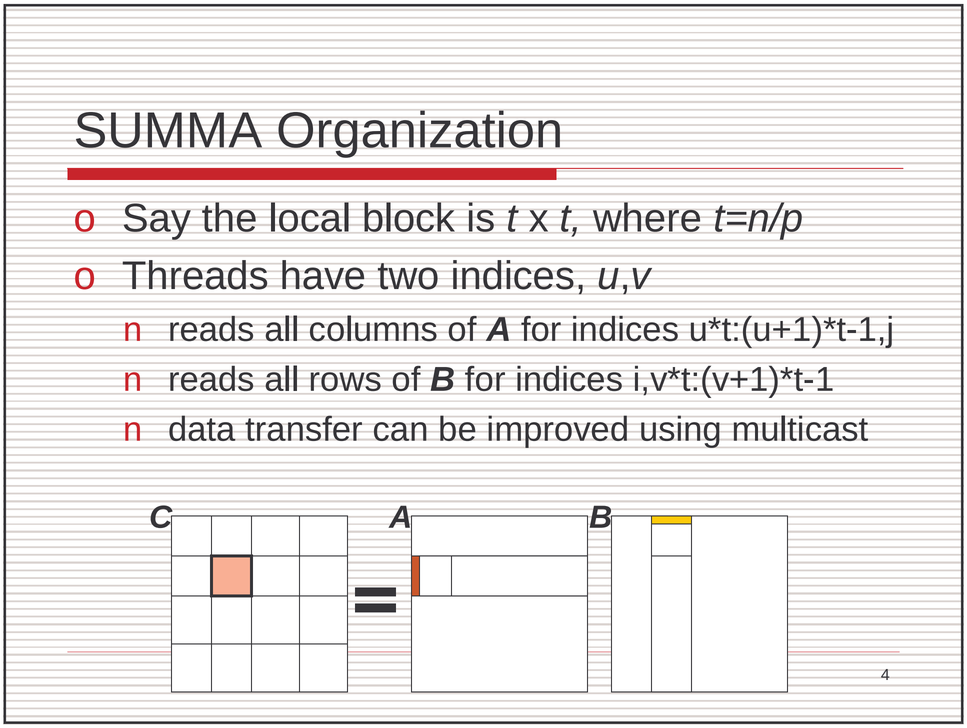
- Homework for 2 weeks returned next time
- New iteration of Chapter 6 handed out
- Homework assigned at end of class
- Project descriptions: Questions?
  - Focus on parallelism of the problem
  - Problem w/little interproc comm won’t work well
  - Huge data file I/O could dominate || comp time
  - Key for me is the transition between P1 and P2

There is still time for revisions



Discuss SUMMA Solution

- Based on email about the homework …
  - Matrices ***A***, ***B*** and ***C*** are assigned to the processors in blocks of size *t* x *t* (or less)
  - Size is *n* x *n*, $p=P^{1/2}$, the allocation is *n/p* x *n/p*
  - How much data is needed to compute block?
  - How much data is needed to do useful work?



SUMMA Organization

- Say the local block is *t* x *t,* where *t=n/p*
- Threads have two indices, *u*,*v*
  - reads all columns of ***A*** for indices u*t:(u+1)*t-1,j
  - reads all rows of ***B*** for indices i,v*t:(v+1)*t-1
  - data transfer can be improved using multicast

SUMMA Preamble Code

```
double A_G[n][n], B_G[n][n], C_G[n][n];
int p=sqrt(P), t=n/p;          Define constants
forall u,v in (0..p-1,0..p-1){ Thread in 2D
  double C[t][t]=is_local(C_G) Ref local tile
  int i,j,k; double a[t], b[t];
  for (i=0;i<t;i++) {
   for (j=0;j<t;j++) {
        C[i][j] = 0.0;         Initialize C
   }
  }
```

Inner Loop of SUMMA

```
for (k=0;k<n;k++) {       For cols of A & rows of B
  a[0:t-1] = A_G[u*t:u*t+t-1,k]; Get col k of A
  b[0:t-1] = B_G[k,v*t:v*t+t-1]; Get row k of B
  for (i=0;i<t;i++) {
   for (j=0;j<t;j++) {
      c[i][j] += a[i]*b[j];    Figure kth terms
   }
  }
 }
}
```

Summary of SUMMA

- Facts:
  - vdG & W advocate blocking for msg passing
  - Works for ***A*** being *m* x *n* and ***B*** being *n* x *p*
  - Works fine when local region is not square
  - Load is balanced esp. of Ceiling/Floor is used
  - Fastest machine independent MM algorithm!
- Key algorithm for 524: Reconceptualizes MM to handle high λ, balance work, use BW well, exploit efficiencies like multicast, …

Bitonic Sort

- One more example of reconceptualizing
- Bitonic sort is a derivative of Ken Batcher's bitonic sorting network
- Key ideas
  - Operations are generally equal amount of work
  - Data motion is carefully controlled

Definitions and Concepts

- A sequence of ordered objects is bitonic if it contains two subsequences, one monotonically non-decreasing and the other monotonically non-increasing, i.e. V or Λ
- Say “increasing” and “decreasing” for short
- Facts:
  - A sorted sequence is bitonic
  - Can rotate a bitonic sequence to get another

An Amazing Property

- The key property of bitonic sequences:

Let A be a bitonic sequence of length 2n
We can divide A into two halves [0,n) & [n,2n) s.t.
- **Each half is bitonic**
- **Every element in [0,n) is <= to each element in [n,2n)**

- What’s the process? Bitonic merge:

```
for (i=0;i<n;i++) {
  if (A[i]>A[i+n]) exch(i,i+n);
}
```

We sort only sequences that are powers of 2

Example

- Watch a bitonic merge
  - Initial: 2 4 6 8 9 5 3 1
  - Divide into two halves: 2 4 6 8 | 9 5 3 1
  - Merging: time →

2 4 6 8	2 4 6 8	2 4 6 8	2 4 3 8	2 4 3 1
9 5 3 1	9 5 3 1	9 5 3 1	9 5 6 1	9 5 6 8

- Bi-merge gets large elements to high end or vice versa

Bitonic Sort

- Abstractly, the bitonic sorting algorithm is
  - Divide the sequence into two halves
  - Sort lower half in ascending order; upper half into descending order
  - Perform bitonic merge on the two halves
  - Recursively bitonically-merge each half until elements are sorted

  - Watch an animation

Thanks to Thomas W. Christopher, Boulder CO

Bitonic As A Parallel Algorithm

- Need to “reverse” the recursive logic, going from bottom up
- Postulate 8 processors with indices:

  bit 0

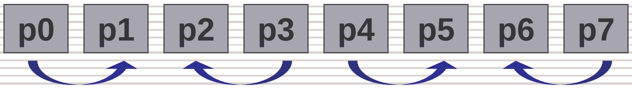
  0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111
  - Processor index guides the sort
- Begin by sorting local vals up/down w/Qsort
- Postulate two merge routines:
  - mergeUp moves smaller items to left half
  - mergeDown moves smaller items to right half

Bitonic Sort, Core Logic

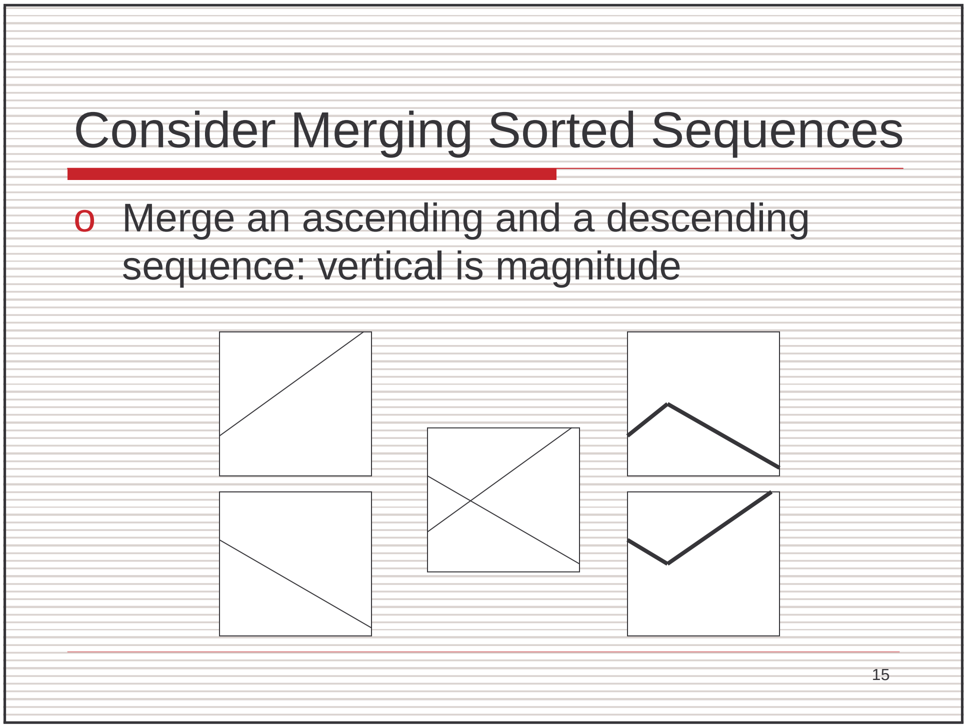
- Processor’s binary no. guides sort/merge

0000 ⇔ 0001	ascending
0010 ⇔ 0011	descending
0100 ⇔ 0101	ascending
0110 ⇔ 0111	descending

- Sort pairs w/ bit 0 different, in bit 1 direction



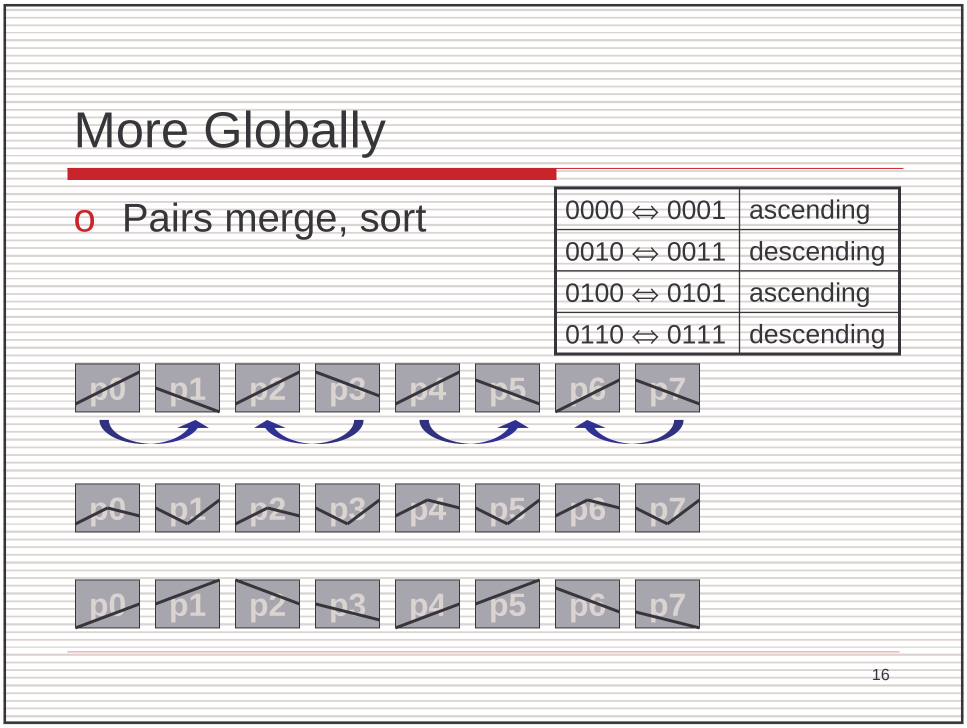
  - Bitonic merge, then internally sort



Consider Merging Sorted Sequences

- Merge an ascending and a descending sequence: vertical is magnitude

---



More Globally

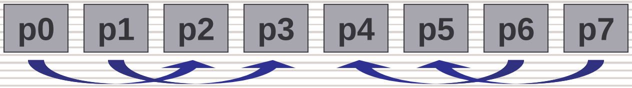
- Pairs merge, sort

0000 ⇔ 0001	ascending
0010 ⇔ 0011	descending
0100 ⇔ 0101	ascending
0110 ⇔ 0111	descending

Repeat At Larger Grain

- Next level exchange

0000 ⇔ 0010	ascending
0001 ⇔ 0011	ascending
0100 ⇔ 0110	descending
0101 ⇔ 0111	descending



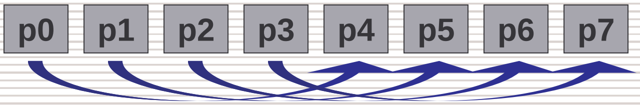
- Recursion

0000 ⇔ 0001	ascending
0010 ⇔ 0011	ascending
0100 ⇔ 0101	descending
0110 ⇔ 0111	descending

Final Round

- Top level exchanges

0000 ⇔ 0100	ascending
0001 ⇔ 0101	ascending
0010 ⇔ 0110	ascending
0011 ⇔ 0111	ascending



- Two second level mergeUps
- Four third level mergeUps
  - Watch an animation

Bitonic Sort

- Many strengths
  - Seriously parallel, all processors work all the time
  - Focuses on concurrent local operations
  - Balances work
  - Generally synchronous, but not in lock step
  - Communication predictable
- Weaknesses
  - Moves data quite a bit

Break

Generalized Reduce and Scan

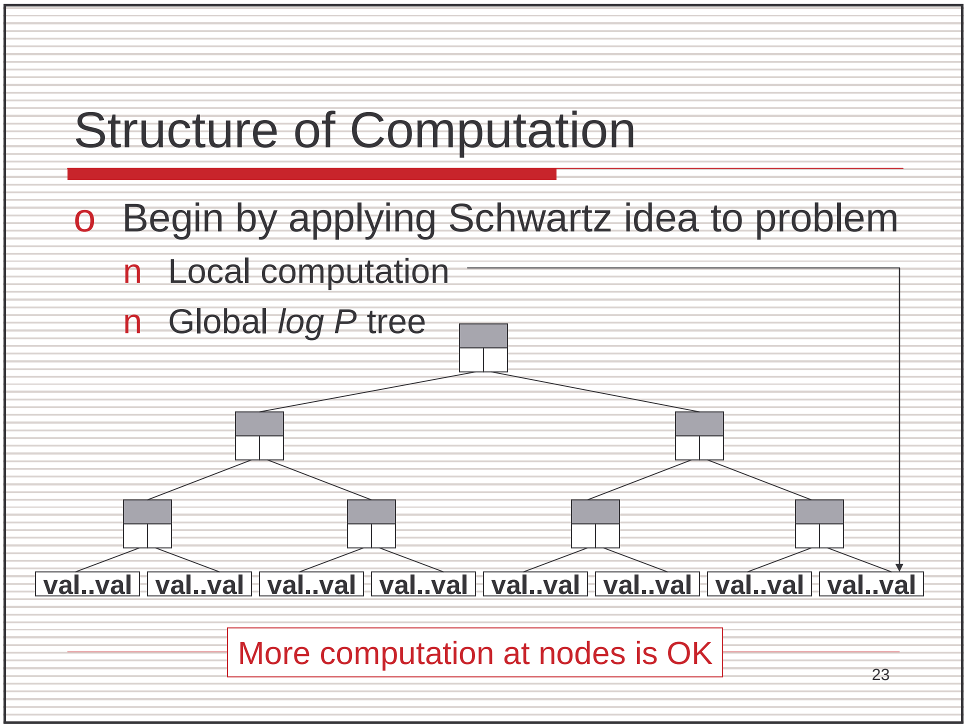
- The importance of reduce/scan has been repeated so often, it is by now our mantra
- In nearly all languages the only available operators are `+`, `*`, `min`, `max`, `&&`, `||`
- The concepts apply much more broadly
- Goal: Understand how to make user-defined variants of reduce/scan specialized to specific situations

Seemingly sequential looping code can be UD-scan

Examples

- Reduce
  - Second smallest, or generally, kth smallest
  - Histogram, counts items in k buckets
  - Length of longest run of value 1s
  - Index of first occurrence of x
- Scan
  - Team standings
  - Find the longest sequence of 1s
  - Last occurrence

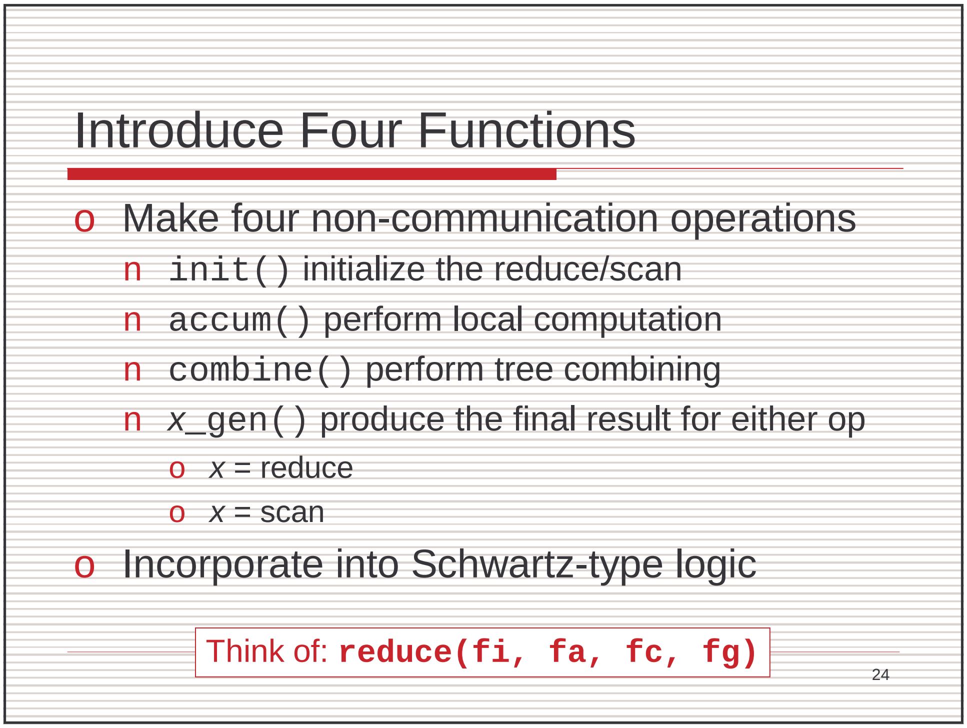
Associativity, but not commutativity, is key



Structure of Computation

- Begin by applying Schwartz idea to problem
  - Local computation
  - Global *log P* tree

More computation at nodes is OK



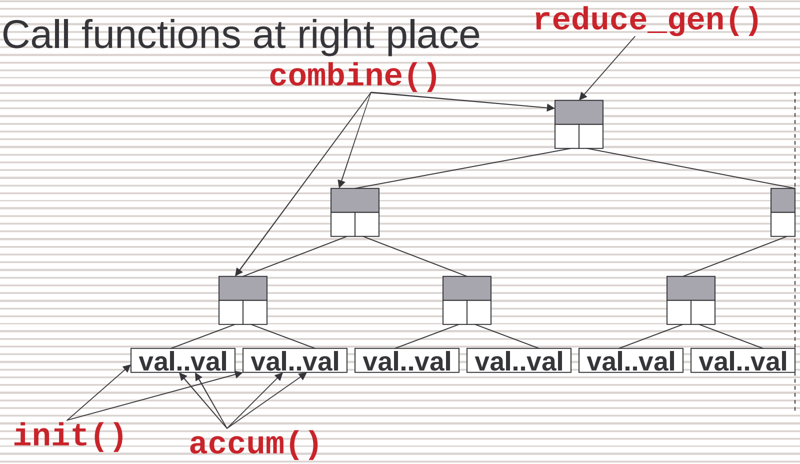
Introduce Four Functions

- Make four non-communication operations
  - `init()` initialize the reduce/scan
  - `accum()` perform local computation
  - `combine()` perform tree combining
  - *x*`_gen()` produce the final result for either op
    - *x* = reduce
    - *x* = scan
- Incorporate into Schwartz-type logic

Think of: **`reduce(fi, fa, fc, fg)`**

Assignment of functions

- Call functions at right place

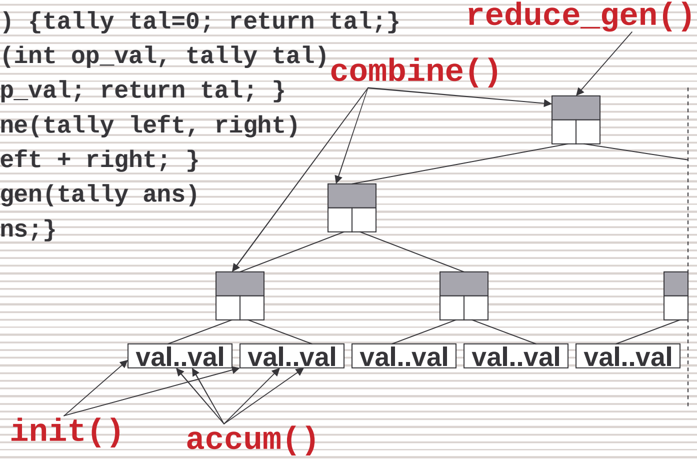


Example: +<<A Definitions

- Sum reduce uses a temporary value known as `tally` to hold items during processing
- Four reduce functions:
  - `tally init() {tally tal=0; return tal;}`
  - `tally accum(int op_val, tally tal)`
    `{tal += op_val; return tal; }`
  - `tally combine(tally left, right)`
    `{return left + right; }`
  - `int reduce_gen(tally ans) {return ans;}`

Details for +<<A

```
tally init() {tally tal=0; return tal;}
tally accum(int op_val, tally tal)
  {tal += op_val; return tal; }
tally combine(tally left, right)
  {return left + right; }
int reduce_gen(tally ans)
  {return ans;}
```



reduce_gen()

combine()

val..val val..val val..val val..val val..val

init() accum()

More Involved Case

- Consider Second Smallest -- useful, perhaps for finding smallest nonzero among non-negative values
- `tally` is a struct of the smallest and next smallest found so far {float sm, nsm}
- Four functions:

```
tally init() {
   tally pair.sm = maxFloat;
   pair.nsm = maxFloat;
   return pair; }
```

Second Smallest (Continued)

- Accumulate

```
tally accum(float op_val, tally tal) {
   if (op_val < tal.sm) {
      tal.nsm = tal.sm;
      tal.sm = op_val;
   } else {
      if (op_val > tal.sm && op_val < tal.nsm)
        tal.nsm = op_val;
     }
   return tal;
}
```

Second Smallest (Continued)

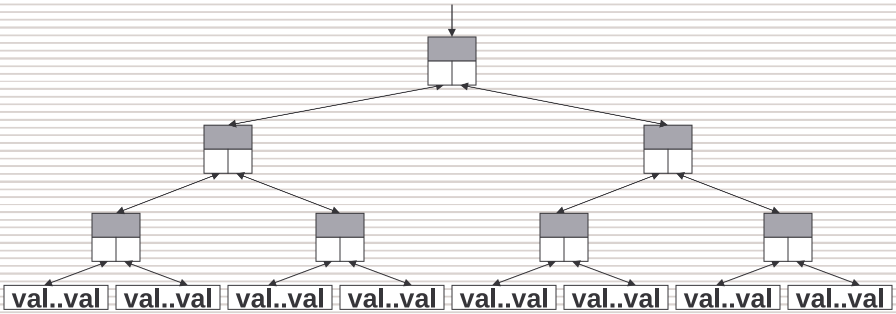
```
tally combine(tally left, right){
   accum(left.sm, right);
   accum(left.nsm, right);
   return right;}
reduce_gen(tally ans) { return ans.nsm;}
```

- Notice that the signatures are all different
- Conceptually easy to write equivalent code, but reduction abstraction clarifies

Generalize to 10th smallest

User-Defined Scan

- Consider operations after the reduce is over
- Consider where functions used: i, a, c, sg



The basic scan logic applies functions

Index of Last Occurrence of x

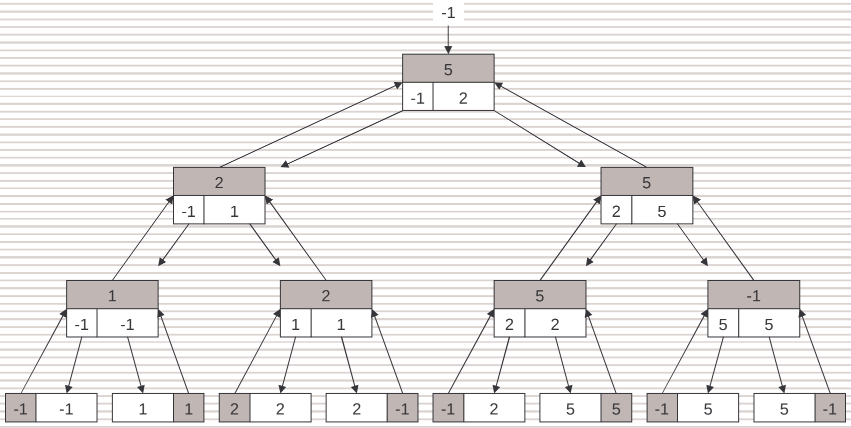
- Assume 0-origin indexing
- `tally` is simply an integer

```
tally init() {
   tally idx = -1;
   return idx;
}
tally accum(int op_val, tally tal, int x, idx) {
   if (op_val == x)
        tal = idx;
   return tal;
}
```

Last Index (Continued)

```
tally combine(tally left, right) {
   if (left > right)
      return left;
   else
      return right;
}
int scan_gen(int op_val, tally ans, int x, idx) {
   if (op_val == x){
       ans = idx;
       return idx;
   } else
     return ans;
}
```

Example x == 1



0 1 1 0 0 1 0 0

-1 1 2 2 2 5 5 5

UD-Scan Summary

- User-defined scan extends UD-reduce
- The operations are essentially the same
  - Applied in additional places
  - Applied with additional arguments
- UD-scan is efficient and powerful … if the language you're writing in doesn't have it, define your own

To think of scanning takes practice

More Generally: UD-Vector Ops

- Scan maintains "context" allowing ordered operations, but that is often not needed
- Vector operations focus on performing some operation across the elements that has global meaning -- longest run of 1s
  - Like all ||ism, the key is formulating local computation so it can be combined to achieve a global result
  - The "scan driver" probably suffices

Blelloch: vectors a sufficient programming model

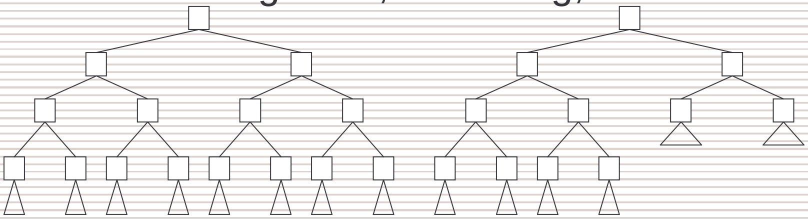
Tree Algorithms

- Trees are an important component of computing
  - The “Schwartz tree” has been logical
  - Trees as data structures are complicated because they are typically more dynamic
  - Pointers are generally not available
  - Work well with work queue approach
  - As usual, we try to exploit locality and minimize communication

---

Breadth-first Trees

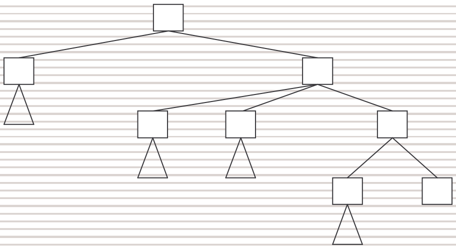
- Common in games, searching, etc



- Split: Pass 1/2 to other processor, continue
  - Stop when processors exhausted
  - Responsible for tree that remains
  - Ideal when work is localized

Depth-first

- Common in graph algorithms



- Get descendants, take one and assign others to the task queue

Key issue is managing the algorithm’s progress

Coordination Among Nodes

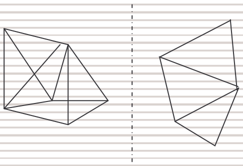
- Tree algorithms often need to know how others are progressing
  - Interrupt works if it is just a search: Eureka!!
  - Record α-β cut-offs in global variable
  - Other pruning data, e.g. best so far, also global
  - Classic error is to consult global too frequently
- Rethink: Is tree data structure essential?

Write essay: Dijkstra’s algorithm is not a good… :)

Complications

- If coordination becomes too involved, consider alternate strategies:

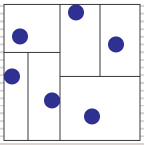
  Graph traverse => local traverse of partitioned graph



- Local computation uses sequential tree algorithms directly … stitch together

Full Enumeration

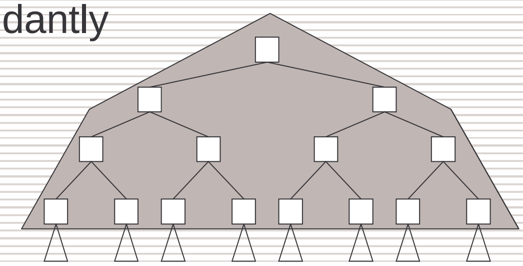
- Trees are a useful data structure for recording spatial relationships: K-D trees



- Generally, decomposition is unnecessary “all the way down” -- but optimization implies two different protocols

Cap Reduces Communication

- The nodes near root can be stored redundantly



  - Processors consult local copy -- alter others to changes

Summary of Parallel Algorithms

- Reconceptualizing is often most effective
- Focus has not be on ||ism, but on other stuff
  - Exploiting locality
  - Balancing work
  - Reducing inter-thread dependences
- We produced general purpose solution mechanisms: UD-reduce and UD-scan
- We like trees, but recognize that direct application is not likely

Discussion

- Next week we start actual programming … what computations have we not considered?

Homework

- Reading: Chapters 6 and 7
  - Read, but do not study …
  - Goal is to conceptualize the two styles
- Write a user-defined scan (4 functions) to perform a new operation of your choice
  - Turn in:
    - Verbal description of the computation
    - Code for 4 functions
    - Small, by hand, example