

Chapel: A Versatile Tool for Teaching Undergraduates Parallel Programming

David P. Bunde, Knox College

Kyle Burke, Colby College

Acknowledgements

- Material drawn from tutorials created with contributions from Johnathan Ebbers, Maxwell Galloway-Carson, Michael Graf, Ernest Heyder, Sung Joo Lee, Andrei Papancea, and Casey Samoore
- Incorporates suggestions from Michael Ferguson
- Work partially supported by NSF awards DUE-1044299 and CCF-0915805. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation



Topics

- Introduction to Chapel
- Chapel in Programming Languages
- Hands-on time
- Chapel in Analysis of Algorithms
- Chapel in Parallel Programming
- Hands-on time
- Final Discussion

Basic Facts about Chapel

- Parallel programming language developed with programmer productivity in mind
- Originally Cray's project under DARPA's High Productivity Computing Systems program
- Suitable for shared- or distributed memory systems
- Installs easily on Linux and Mac OS; use Cygwin to install on Windows

Why Chapel?

- Flexible syntax; only need to teach features that you need
- Provides high-level operations
- Designed with parallelism in mind

Flexible Syntax

- Supports scripting-like programs:
 `writeln("Hello World!");`
- Also provides objects and modules

Provides High-level Operations

- Reductions

Ex: `x = + reduce A` //sets x to sum of elements of A

Also valid for other operators (min, max, *, ...)

- Scans

Like a reduction, but computes value for each prefix

`A = [1, 3, 2, 5];`

`B = + scan A;` //sets B to [1, 1+3=4, 4+2=6, 6+5=11]

Provides High-level Operations (2)

- Function promotion:

$B = f(A);$ //applies f elementwise for any function f

- Includes built-in operators:

$C = A + 1;$

$D = A + B;$

$E = A * B;$

...

Designed with Parallelism in Mind

- Operations on previous slides parallelized automatically
- Create asynchronous task w/ single keyword
- Built-in synchronization for tasks and variables

Your Presenters are...

- Enthusiastic Chapel users
- Interested in high-level parallel programming
- Educators who use Chapel with students
- **NOT connected to Chapel development team**

Chapel Resources

- Materials for this workshop
<http://faculty.knox.edu/dbunde/teaching/chapel/SIGCSE14/>
- Our tutorials
<http://faculty.knox.edu/dbunde/teaching/chapel/>
<http://cs.colby.edu/kgburke/?resource=chapelTutorial>
- Chapel website (tutorials, papers, language specification)
<http://chapel.cray.com>
- Mailing lists (on SourceForge)

Practice Systems

- We have practice accounts set up for use during the workshop
- Get handout from one of the instructors
- Will keep accounts available for a couple of weeks

“Hello World” in Chapel

- Create file hello.chpl containing
`writeln(“Hello World!”);`
- Compile with
`chpl -o hello hello.chpl`
- Run with
`./hello`

Variables and Constants

- Variable declaration format:
[config] var/const identifier : type;

var x : int;

const pi : real = 3.14;

config const numSides : int = 4;

Serial Control Structures

- if statements, while loops, and do-while loops are all pretty standard
- Difference: Statement bodies must either use braces or an extra keyword:

if(x == 5) **then** y = 3; else y = 1;

while(x < 5) **do** x++;

Example: Reading until eof

```
var x : int;  
while stdin.read(x) {  
    writeln("Read value ", x);  
}
```


Procedures/Functions

arg_type argument omit for generic function

```
proc addOne(in val : int, inout val2 : int) : int {  
    val2 = val + 1;  
    return val + 1;  
}
```

return type
(omit if none
or if can be inferred)

Arrays

- Indices determined by a range:
var A : [1..5] int; //declares A as array of 5 ints
var B : [-3..3] int; //has indices -3 thru 3
var C : [1..10, 1..10] int; //multi-dimensional array
- Accessing individual cells:
A[1] = A[2] + 23;
- Arrays have runtime bounds checking

For Loops

- Ranges also used in for loops:
for i in 1..10 do statement;
for i in 1..10 {
 loop body
}
- Can also use array or anything iterable

Timing code

```
use Time;                                //include Time library
```

```
var timer = new Timer();                 //create Timer object
```

```
timer.start();
```

```
//do something...
```

```
timer.stop();
```

```
timer.elapsed() //returns (real-valued) number of seconds
```

```
timer.clear(); //get ready to use it again!
```

Programming Languages

- High-Performance Computing as Paradigm
- Lots of design choices in Chapel to discuss:
 - Task Creation (instead of Threads) with 'begin'.
 - Task Synchronicity with 'sync' and cobegin
 - Parallel loops: forall and coforall
 - Thread safety using variable 'sync'
 - reduce overcomes bottleneck

PL: Task Generation

```
var total = 0;  
for i in 1..100 do total += i;  
  
writeln('Sum is ', total, '.');
```

We can add a Timer to measure running time!

PL: Task Generation

```
var total = 0;
for i in 1..100 do total += i;

writeln('Sum is ', total, '.');
```

We can add a Timer to measure running time!

```
use Time;
var timer: Timer;
var total = 0;
timer.start();
for i in 1..100 do total += i;
timer.stop();

writeln('Sum is ', total, '.');
writeln('That took ', timer.elapsed(), ' seconds.');
```

PL: Task Generation

Now let's use another thread!

```
use Time;
var timer: Timer;
var total = 0;
var highTotal = 0;
var lowTotal = 0;
timer.start();
begin ref(highTotal) {
    for i in 51..100 do highTotal += i;
}
for i in 1..50 do lowTotal += i;
total = lowTotal + highTotal;
timer.stop();

writeln('Sum is ', total, '.');
writeln('That took ', timer.elapsed(), ' seconds.');
```

Note: `ref(highTotal)` at begin

PL: Task Generation

Now let's use another thread!

```
use Time;
var timer: Timer;
var total = 0;
var highTotal = 0;
var lowTotal = 0;
timer.start();
begin ref(highTotal) {
    for i in 51..100 do highTotal += i;
}
for i in 1..50 do lowTotal += i;
total = lowTotal + highTotal;
timer.stop();

writeln('Sum is ', total, '.');
writeln('That took ', timer.elapsed(), ' seconds.');
```

Result: faster, but sometimes incorrect.

PL: Synchronization

Incorrect: top thread may not finish.

Chapel provides a solution: `sync`

```
sync {  
  begin {  
    ...  
  }  
  begin {  
    ...  
  }  
  ...  
}
```

PL: Synchronization

Use sync:

```
...
timer.start();
sync {
    begin ref(highTotal) {
        for i in 51..100 do highTotal += i;
    }
    begin ref(lowTotal) {
        for i in 1..50 do lowTotal += i;
    }
}
total = lowTotal + highTotal;
...
```

PL: Syntactic Sugar

Ask students: How common is this?

```
sync {  
    begin {  
        //single line of code  
    }  
    begin {  
        //another single line  
    }  
    . . .  
    begin {  
        //even yet another single line  
    }  
}
```

So, what did language designers do?

PL: Syntactic Sugar

```
cobegin {  
    //single line of code  
    //another single line  
    . . .  
    //even yet another single line  
}
```

PL: forall

forall: data-parallel loop

```
var sum = 0;
forall i in 1..100 {
    sum += i;
}
writeln("Sum is: ", sum, ".");
```

PL: forall

forall: data-parallel loop

```
var sum = 0;
forall i in 1..100 {
    sum += i;
}
writeln("Sum is: ", sum, ".");
```

Ask: Why doesn't this work?

PL: HPC Concepts

- Why doesn't it work?
 - Race conditions
 - Atomicity
 - Synchronization solutions

PL: forall

One solution: synchronized variables

```
var sum : sync int;  
sum = 0;  
forall i in 1..100 {  
    sum += i;  
}  
writeln("Sum is: ", sum, ".");
```

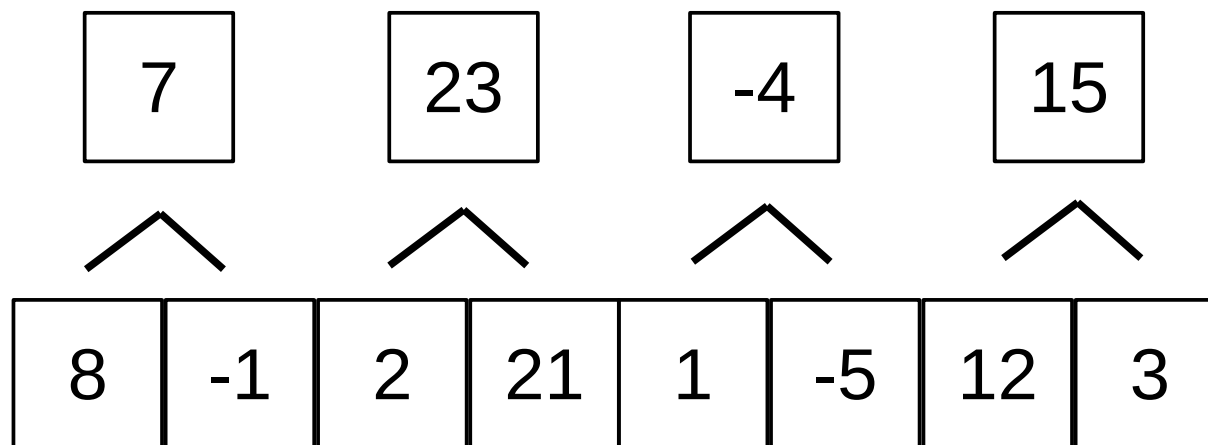
PL: sync bottleneck and reduce

- sync causes a bottleneck:
 - Running time still technically linear.
- Reductions:
 - Divide-and-conquer solution

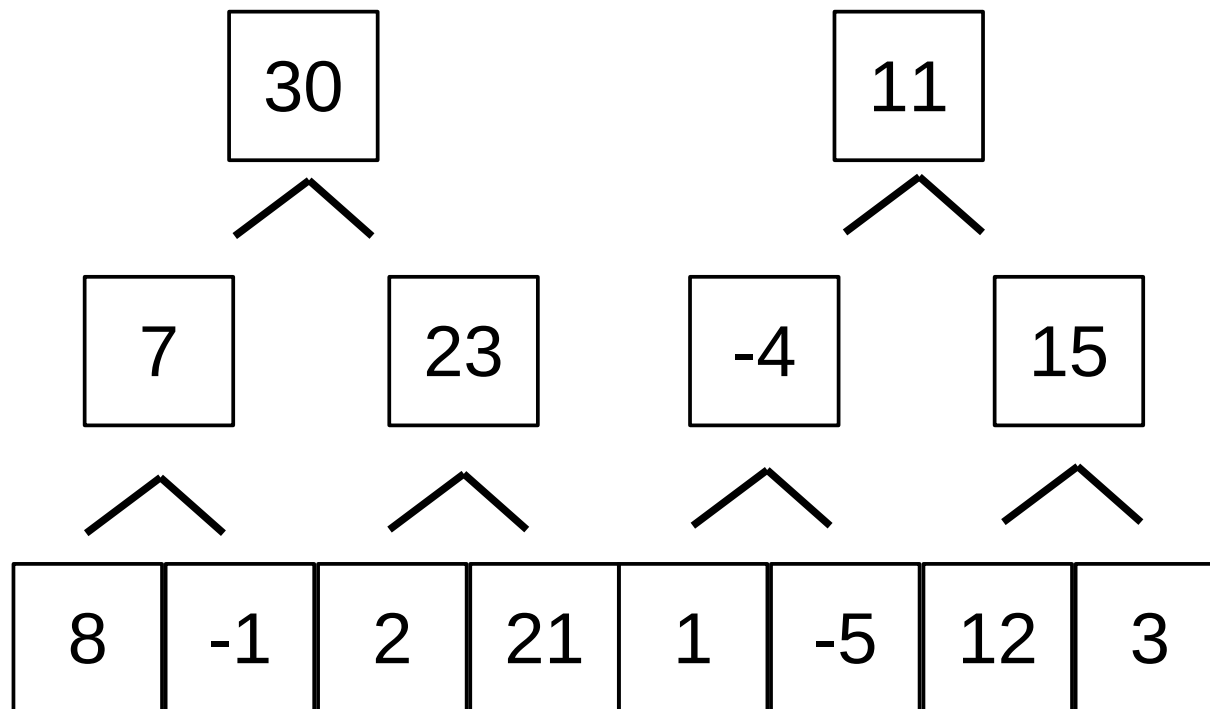
PL: Reductions

8	-1	2	21	1	-5	12	3
---	----	---	----	---	----	----	---

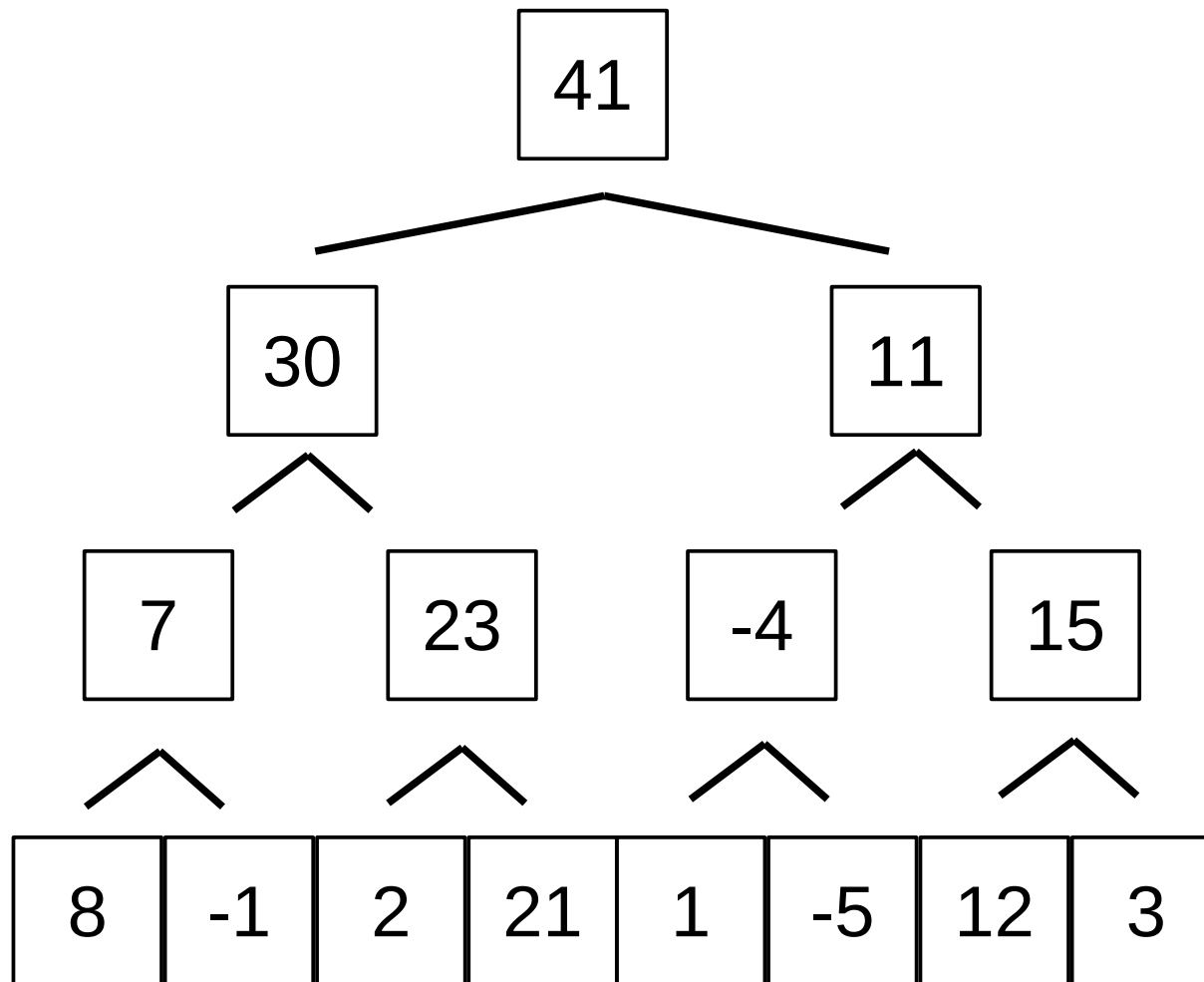
PL: Reductions



PL: Reductions



PL: Reductions



PL: sync bottleneck and reduce

- sync causes a bottleneck:
 - Running time still technically linear.
- Reductions:
 - Divide-and-conquer solution

PL: sync bottleneck and reduce

- sync causes a bottleneck:
 - Running time still technically linear.
- Reductions:
 - Divide-and-conquer solution
 - Simplify with 'reduce' keyword!

PL: sync bottleneck and reduce

```
var integers : [1..100] int;  
forall i in integers.domain {  
    integers[i] = i;  
}  
var sum = + reduce integers;
```

PL: sync bottleneck and reduce

```
var integers : [1..100] int;  
forall i in integers.domain {  
    integers[i] = i;  
}  
var sum = + reduce integers;
```

One line solution?

PL: sync bottleneck and reduce

```
var sum = + reduce (1..100);
```

PL: sync bottleneck and reduce

All intermediate values?

```
var sum = + scan array;
```

8	-1	2	21	1	-5	12	3
---	----	---	----	---	----	----	---

PL: sync bottleneck and reduce

All intermediate values?

```
var sum = + scan array;
```

8	-1	2	21	1	-5	12	3
---	----	---	----	---	----	----	---

8	7	9	30	31	26	38	41
---	---	---	----	----	----	----	----

PL: sync bottleneck and reduce

Fun Uses!

```
var factorials = * scan (1..10);  
  
var threes : [1..10] int;  
forall i in threes.domain do  
    threes[i] = 3;  
var powersOfThree = * scan threes;
```

Chapel Ranges

- What is a range?
- How are ranges used?
- Range operations

Chapel Ranges

- What is a range?
 - A range of values
 - Ex: `var someNaturals : range = 0..50;`
- How are they used?
 - Indexes for Arrays
 - Iteration space in loops
- Are there cool operations?

Chapel Ranges

- What is a range?
 - A range of values
 - Ex: `var someNaturals : range = 0..50;`
- How are they used?
 - Indexes for Arrays
 - Iteration space in loops
- Are there cool operations?

Yes!

Range Operation Examples

```
var someNaturals: range = 0..50;
```

```
var someEvens = someNaturals by 2;
```

```
    (someEvens: 0, 2, 4, ..., 48, 50)
```

```
var someOdds = someEvens align 1;
```

```
    (someOdds: 1, 3, 5, 7, ..., 47, 49)
```

```
var fewerOdds = someOdds # 6;
```

```
    (fewerOdds: 1, 3, 5, 7, 9, 11)
```

Other Cool Range Things

- Can create “infinite” ranges:
var naturals: range = 0..;
- Ranges in the “wrong order” are auto-empty:
var nothing: range = 2..-2;
- Otherwise, negatives are just fine

Chapel Domains

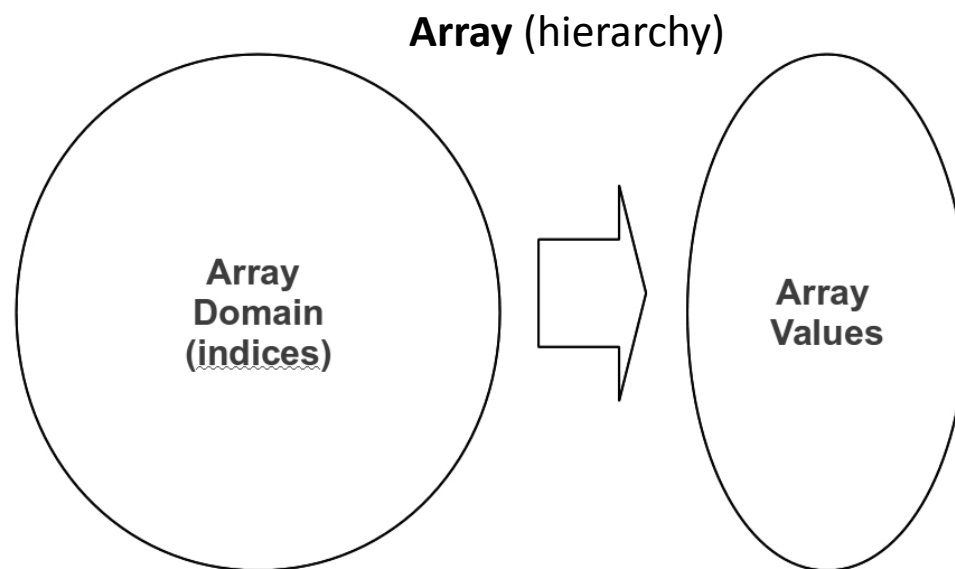
- What is a domain?
- How are domains used?
- Operations on domains
- Example: Game of Life

Chapel Domains

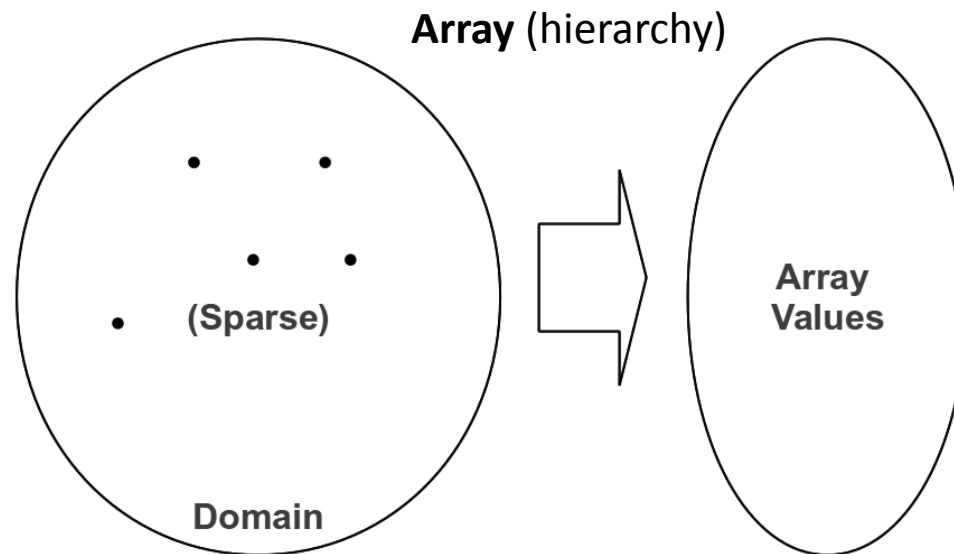
- Domain: index set
 - Used to simplify addressing
 - Every array has a domain to hold its indices
 - Can include ranges or be sparse
- Example:

```
var A: [1..10] int; //indices are 1, 2, ..., 10
...
for i in A.domain {
    //do something with A[i]
}
```

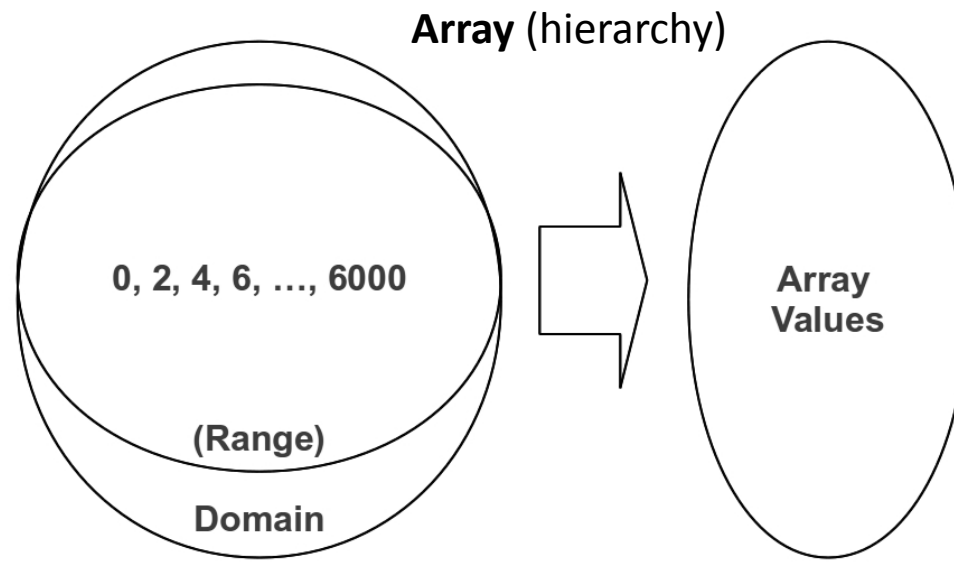
Chapel Domains



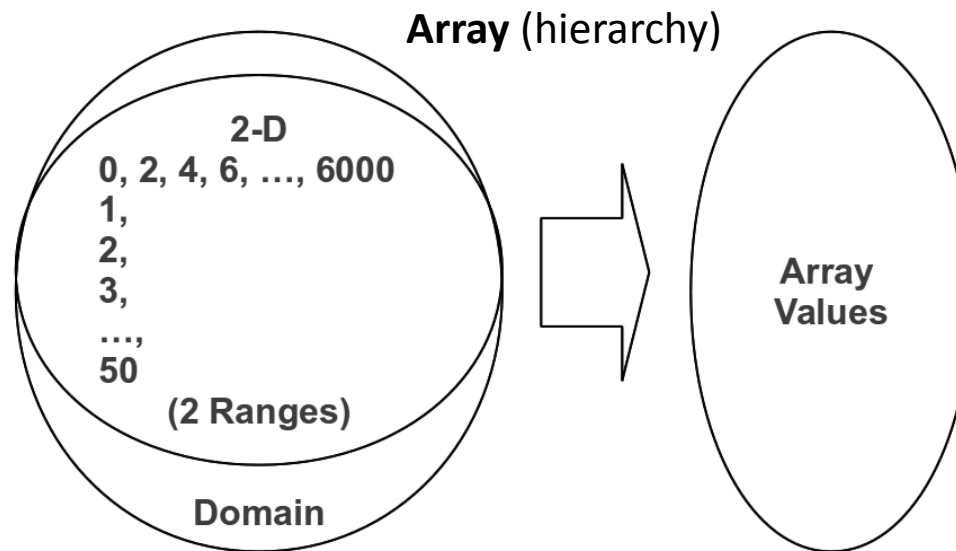
Chapel Domains



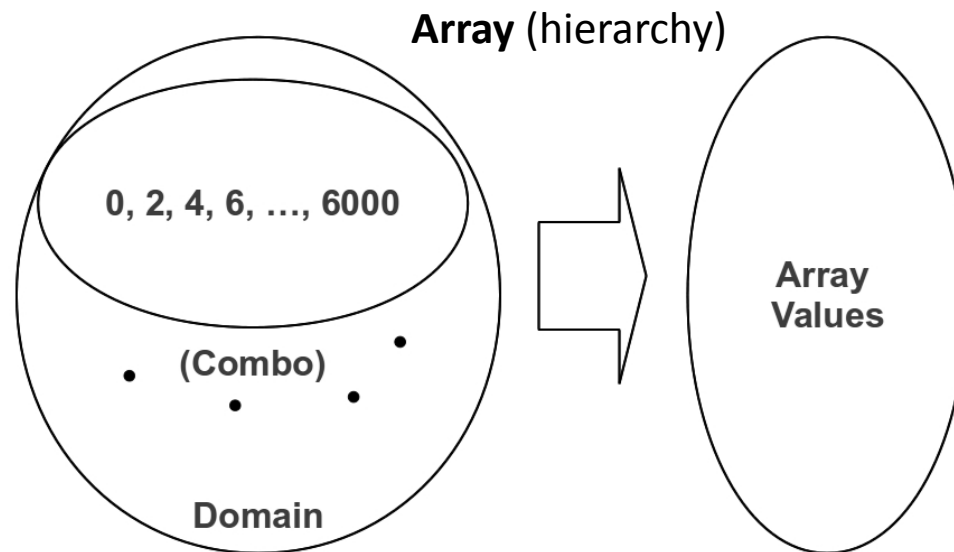
Chapel Domains



Chapel Domains



Chapel Domains



Chapel Domains

- Domain Declaration:
 - var D: domain(2) = {0..m, 0..n};
 - D is 2-D domain with (m+1) x (n+1) entries
 - var A: [D] int;
 - A is an array of integers with D as its domain

Chapel Domains

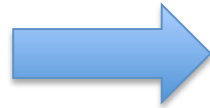
- Domain Declaration:
 - var D: domain(2) = {0..m, 0..n};
 - D is 2-D domain with (m+1) x (n+1) entries
 - var A: [D] int;
 - A is an array of integers with D as its domain

Why is this useful?

Chapel Domains

- Changing D changes A automatically!
- $D = \{1..m, 0..n+1\}$
decrements height; increments width!
(adds zeroes)

1	2	3
4	5	6
7	8	9



4	5	6	0
7	8	9	0

Domains vs. Ranges

- Despite how similar they seem so far, domains and ranges are different
 - Domains remain tied to arrays so that resizing the domain resizes the array:

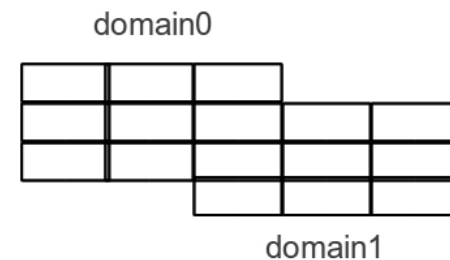
<code>var R : range = 1..10;</code>	<code>var D : domain(1) = {1..10};</code>
<code>var A : [R] int;</code>	<code>var A : [D] int;</code>
<code>R = 0..10; //no effect on array</code>	<code>D = 0..10; //resizes array</code>
<code>A[0] = 5; //runtime error</code>	<code>A[0] = 5; //ok</code>

- Domains are more general; some are not sets of integers

Domain Slices (Intersection)

domain0: {0..2, 1..3}

domain1: {1..3, 3..5}

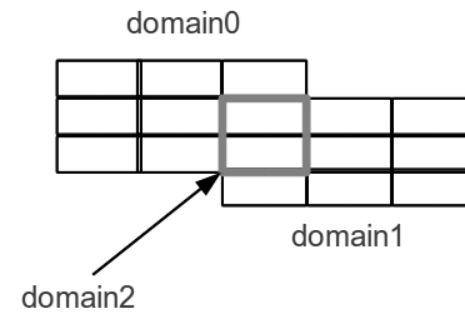


Domain Slices (Intersection)

domain0: {0..2, 1..3}

domain1: {1..3, 3..5}

domain2: {1..2, 3..3}



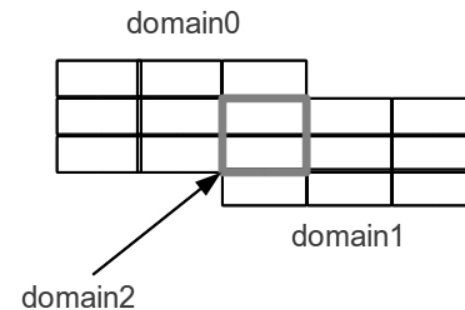
Domain Slices (Intersection)

//domain2 is the intersection of domain1 and domain0
var domain2 = domain1 [domain0];

domain0: {0..2, 1..3}

domain1: {1..3, 3..5}

domain2: {1..2, 3..3}



Domain Slices (Intersection)

//domain2 is the intersection of domain1 and domain0
var domain2 = domain1 [domain0];

PL: Projects

- Matrix Multiplication
 - Matrix-vector multiplication in class
 - Different algorithms:
 - Column-by-column
 - One entry at a time
- Collatz conjecture testing
 - Generate lots of tasks (coforall)
 - How to synchronize?

PL: Takeaways

- Lots of language features to discuss!
- Learning HPC \leftrightarrow Motivates Syntax
- Students love it!

Hands-on time

<http://cs.colby.edu/kgburke/?resource=sigcse>

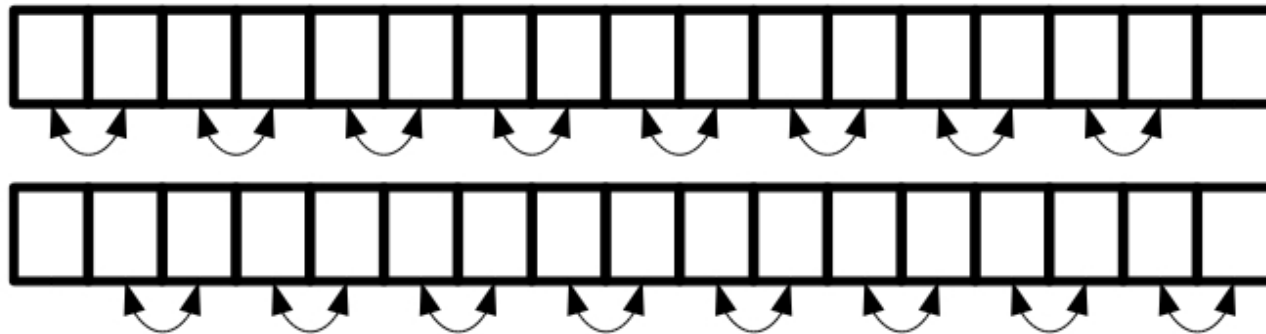
Analysis of Algorithms

- Chapel material
 - Assign basic tutorial
 - Teach forall & cobegin (also algorithmic notation)
- Projects
 - Partition integers
 - BubbleSort
 - MergeSort
 - Nearest Neighbors

Algorithms Project: List Partition

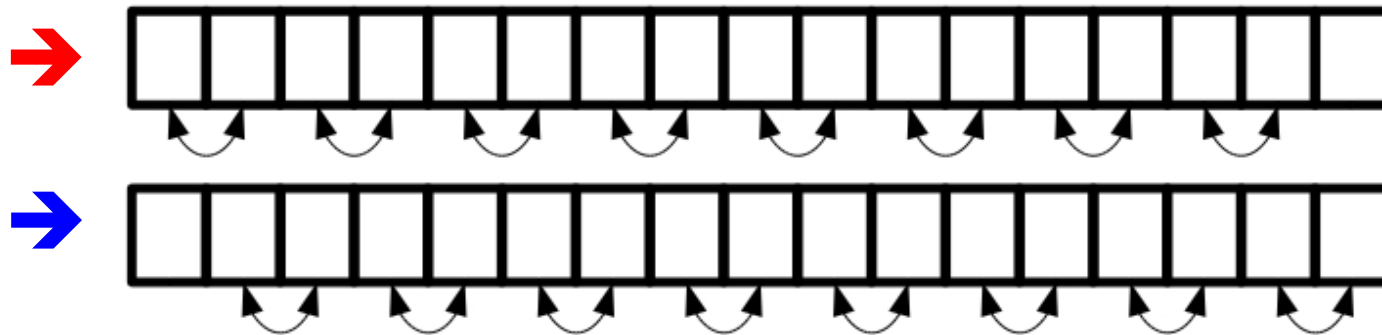
- Partition a list to two equal-summing halves.
- Brute-force algorithm (don't know P vs NP yet)
- Questions:
 - What are longest lists you can test?
 - What about in parallel?
- Trick: enumerate possibilities and use forall

Algorithms Project: BubbleSort



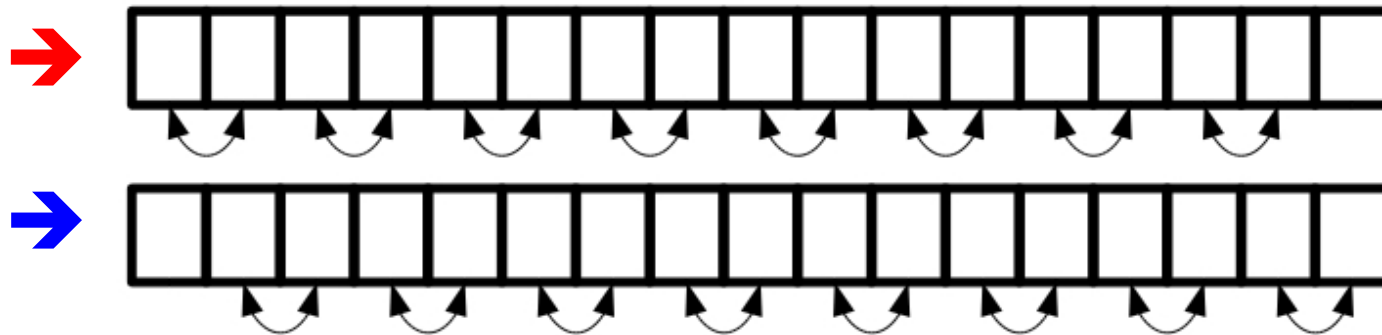
- Instead of left-to-right, test all pairs in two steps!
- Two nested for all loops (in sequence) inside a for loop

Algorithms Project: BubbleSort



```
for i in 0..n-1 {  
    forall k in 0..n/2  
        //compare  $2k$  to  $2k+1$  (maybe swap)  
    forall k in 0..n/2-1  
        //compare  $2k+1$  to  $2k+2$  (maybe swap)  
}
```

Algorithms Project: BubbleSort



```
for i in 0..n-1 {  
    forall k in 0..n/2  
        //compare 2k to 2k+1 (maybe swap)  
    forall k in 0..n/2-1  
        //compare 2k+1 to 2k+2 (maybe swap)  
}
```

$$\lim_{p \rightarrow n} T(n, p) = O(n)$$

Algorithms Project: MergeSort

Parallel divide-and-conquer: use cobegin

12	8	5	15	7	4	4	0	16	7	1	9
----	---	---	----	---	---	---	---	----	---	---	---

12	8	5	15	7	4
----	---	---	----	---	---

4	0	16	7	1	9
---	---	----	---	---	---

Algorithms Project: MergeSort

Parallel divide-and-conquer: use cobegin

12	8	5	15	7	4	4	0	16	7	1	9
----	---	---	----	---	---	---	---	----	---	---	---

4	5	7	8	12	15
---	---	---	---	----	----

0	1	4	7	9	16
---	---	---	---	---	----

Algorithms Project: MergeSort

Parallel divide-and-conquer: use cobegin

12	8	5	15	7	4	4	0	16	7	1	9
----	---	---	----	---	---	---	---	----	---	---	---

4	5	7	8	12	15
---	---	---	---	----	----

0	1	4	7	9	16
---	---	---	---	---	----

0	1	4	4	5	7	7	8	9	12	15	16
---	---	---	---	---	---	---	---	---	----	----	----

Algorithms Project: Nearest Neighbors

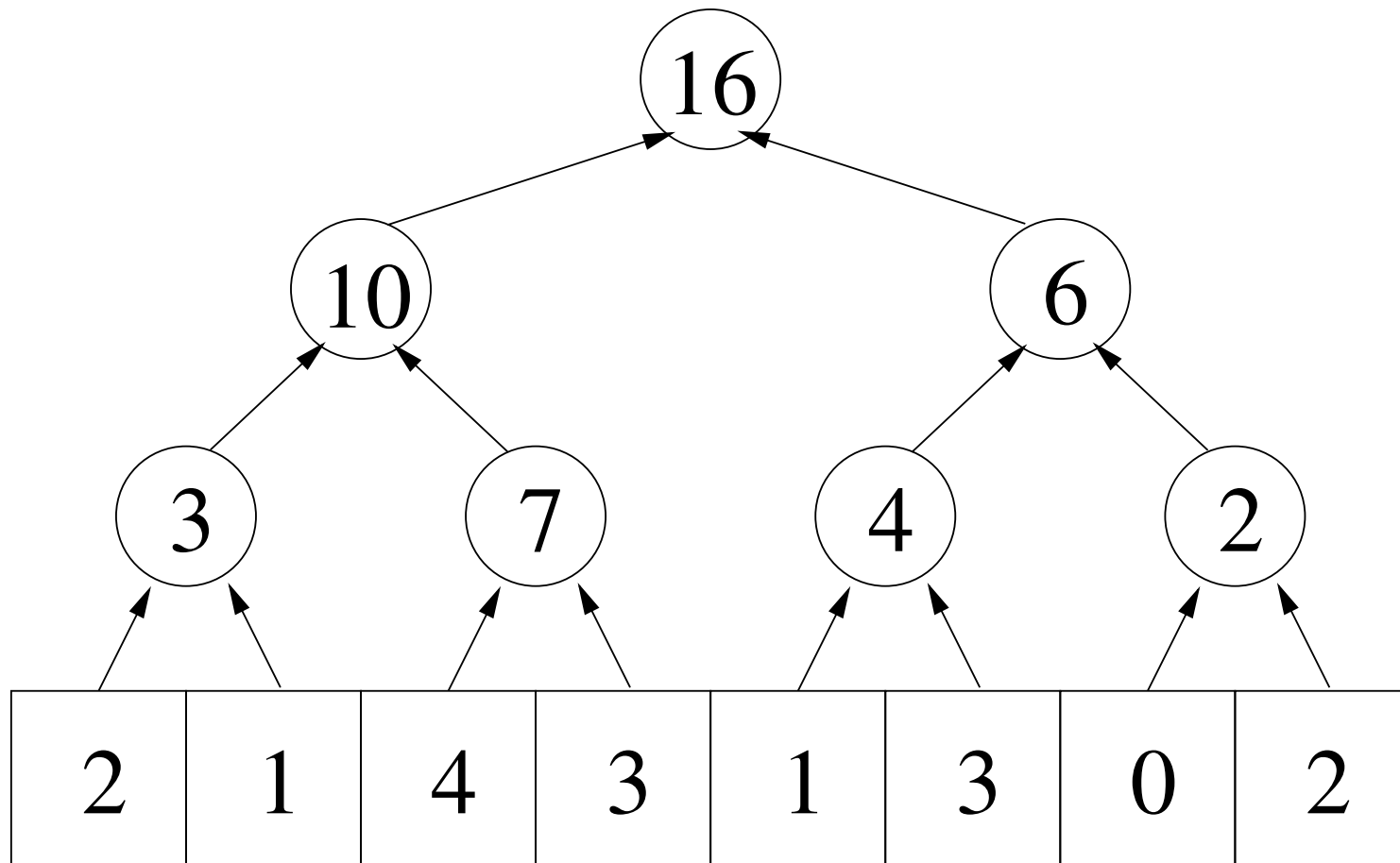
- Find closest pair of (2-D) points.
- Two algorithms:
 - Brute Force
 - (use a forall like bubbleSort)
 - Divide-and-Conquer
 - (use cobegin)
 - A bit tricky
- Value of parallelism: much easier to program the brute-force method

Reductions II: The Revenge

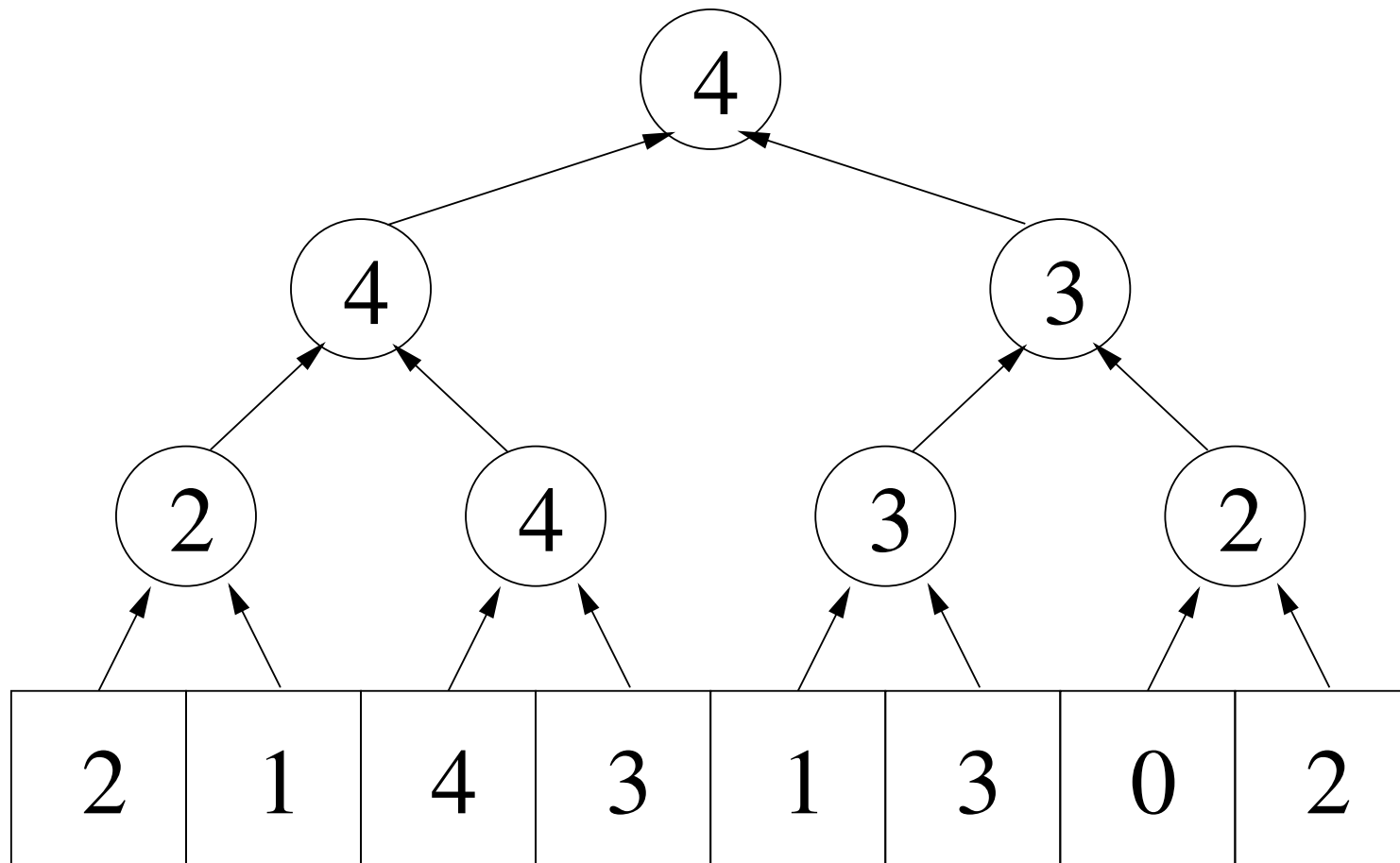
Summing values in an array

2	1	4	3	1	3	0	2
---	---	---	---	---	---	---	---

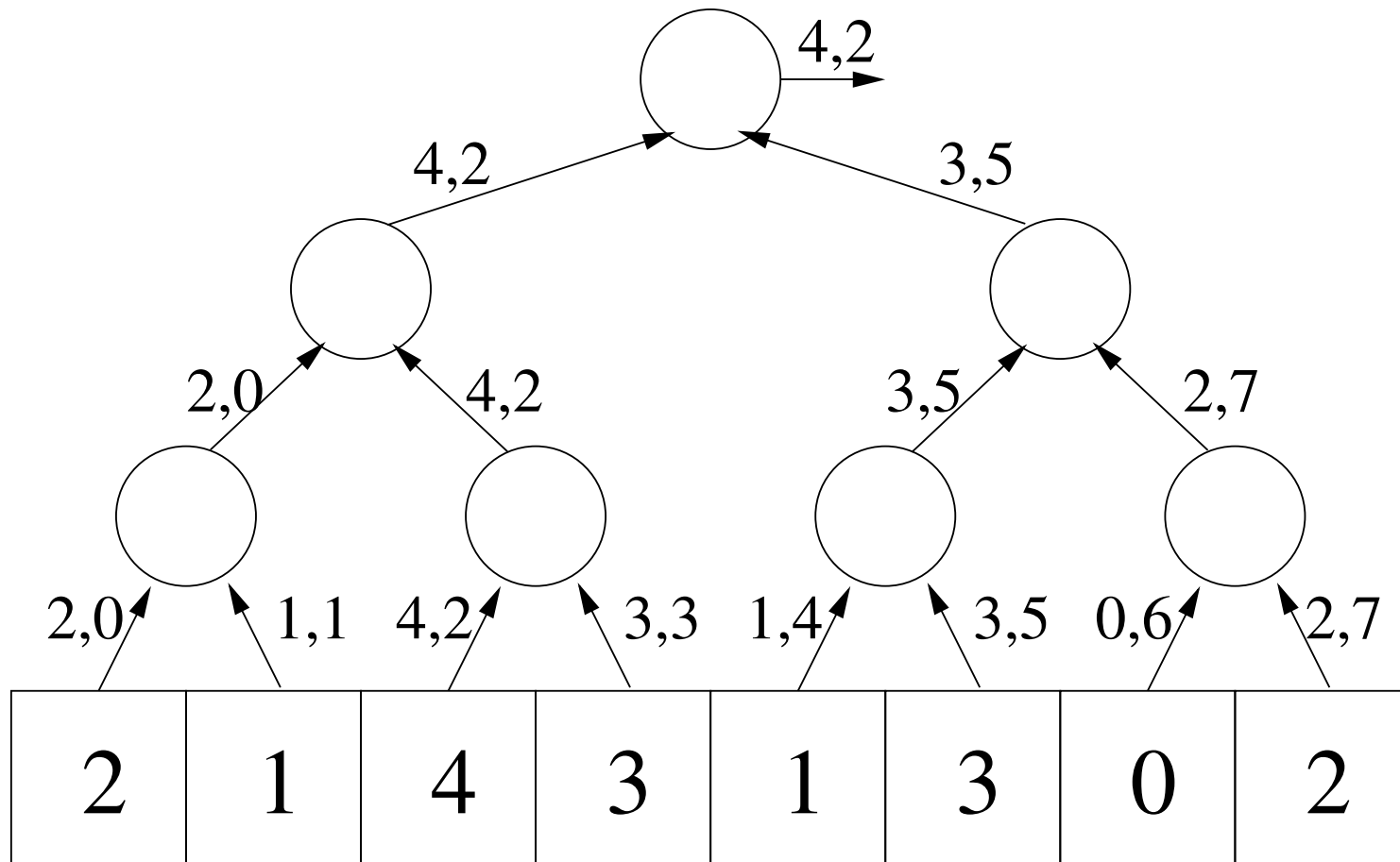
Summing values in an array



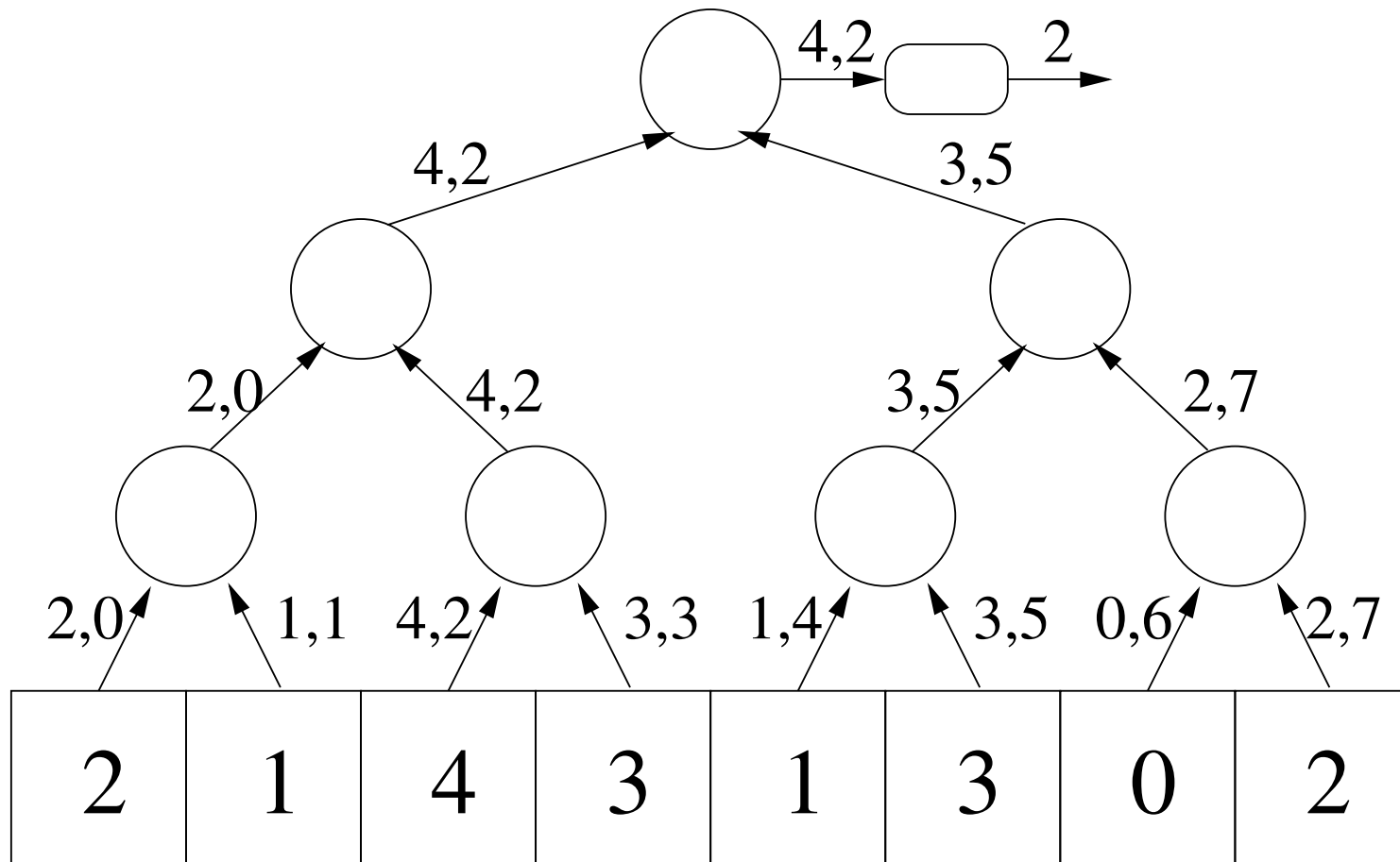
Finding max of an array



Finding the maximum index



Finding the maximum index



Parts of a reduction

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally

Parts of a reduction

- Tally: Intermediate state of computation
(value, index)
- Combine: Combine 2 tallies
take whichever pair has larger value
- Reduce-gen: Generate result from tally
return the index

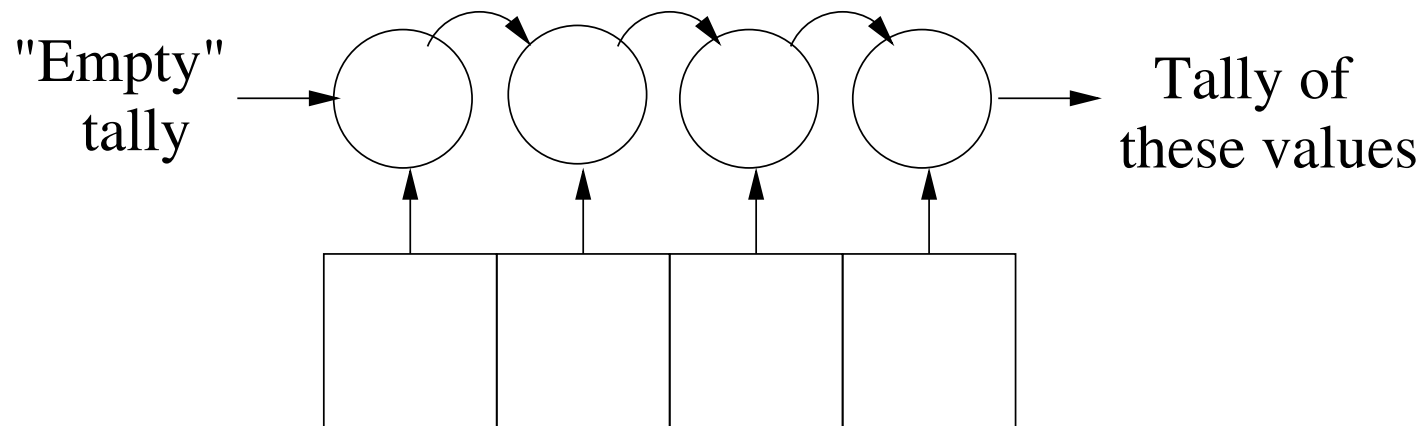
Two issues

- Need to convert initial values into tallies
- May want separate operation for values local to a single processor



Two issues

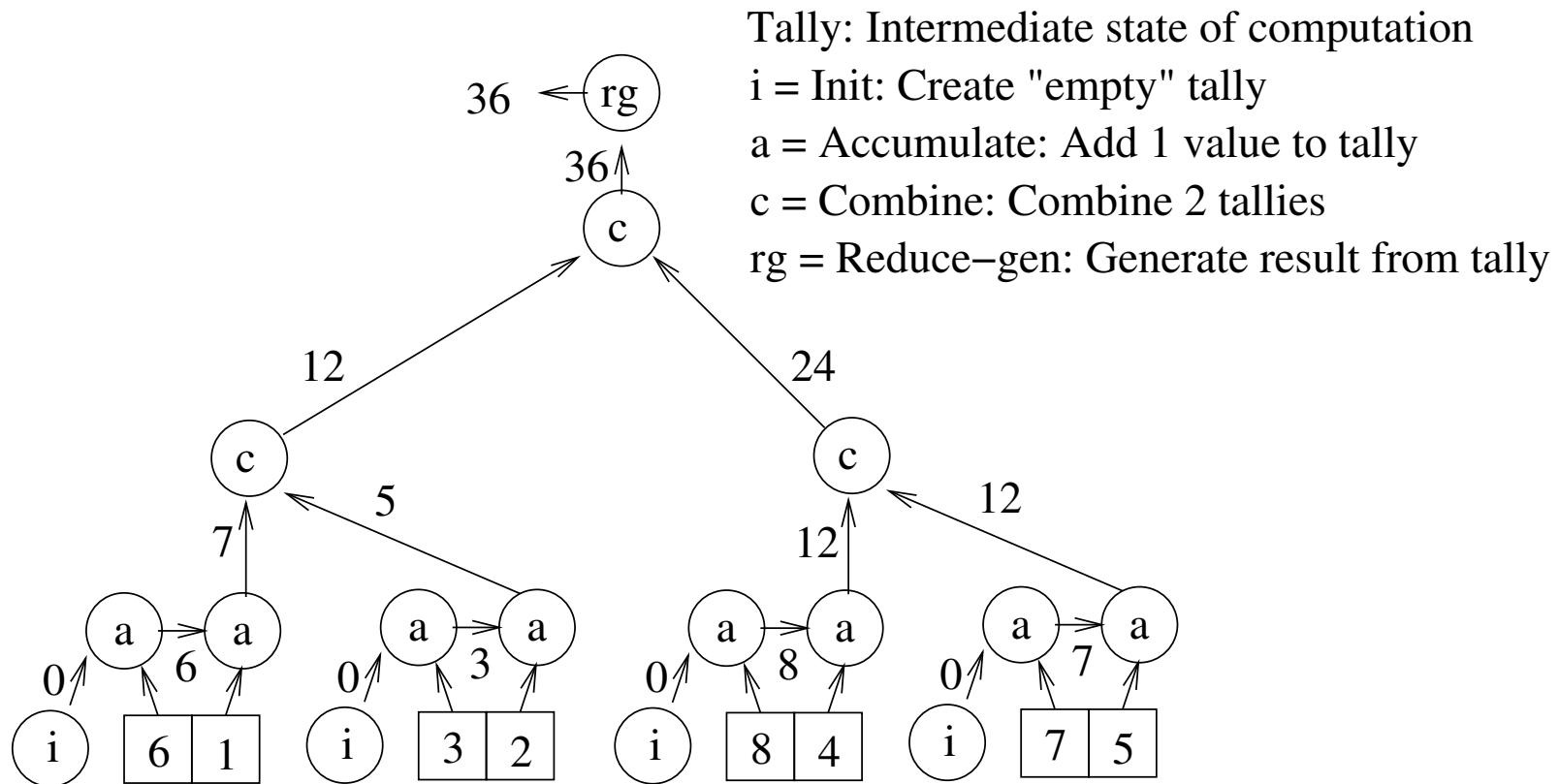
- Need to convert initial values into tallies
- May want separate operation for values local to a single processor



Parts of a reduction

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally

Parallel reduction framework



Defining reductions

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally

Sample problems: +

Defining reductions

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally

Sample problems: +, histogram

Defining reductions

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally

Sample problems: +, histogram, max

Defining reductions

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally

Sample problems: +, histogram, max, 2nd largest

Defining reductions

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally

Sample problems: +, histogram, max, 2nd largest,
length of longest run

Can go beyond these...

- indexOf (find index of first occurrence)
- sequence alignment [Srinivas Aluru]
- n-body problem [Srinivas Aluru]

Relationship to dynamic programming

- Challenges in dynamic programming:
 - What are the table entries?
 - How to compute a table entry from previous entries?
- Challenges in reduction framework:
 - What is the tally?
 - How to compute a new tallies from previous ones?

Reductions in Chapel

- Express reduction operation in single line:
`var s = + reduce A; //A is array, s gets sum`
- Supports +, *, ^ (xor), &&, ||, max, min, ...
- minloc and maxloc return a tuple with value and its index:
`var (val, loc) = minloc reduce A;`

Reduction example

- Can also use reduce on function plus a range
- Ex: Approximate $\pi/2$ using $\int_{-1}^1 \sqrt{1-x^2} dx$:

```
config const numRect = 10000000;  
const width = 2.0 / numRect;           //rectangle width  
const baseX = -1 - width/2;  
const halfPI = + reduce [i in 1..numRect]  
    (width * sqrt(1.0 - (baseX + i*width)**2));
```

Defining a custom reduction

- Create object to represent intermediate state
- Must support
 - accumulate: adds a single element to the state
 - combine: adds another intermediate state
 - generate: converts state object into final output

Classes in Chapel

```
class Circle {  
    var radius : real;  
    proc area() : real {  
        return 3.14 * radius * radius;  
    }  
}
```

```
var c1, c2 : Circle;           //creates 2 Circle references  
c1 = new Circle(10);          /* uses system-supplied constructor  
                                to create a Circle object  
                                and makes c1 refer to it */  
c2 = c1;                       //makes c2 refer to the same object  
delete c1;                     //memory must be manually freed
```

Inheritance

```
class Circle : Shape {    //Circle inherits from Shape
    ...
}
```

```
var s : Shape;
s = new Circle(10.0); //automatic cast to base class
var area = s.area();  /* call recipient determined
                        by object's dynamic type */
```

Example “custom” reduction

```
class MyMin : ReduceScanOp { //finds min element (equiv. to built-in “min”)
    type eltType;                //type of elements
    var soFar : eltType = max(eltType); //minimum so far

    proc accumulate(val : eltType) {
        if(val < soFar) { soFar = val; }
    }

    proc combine(other : MyMin) {
        if(other.soFar < soFar) { soFar = other.soFar; }
    }

    proc generate() { return soFar; }
}
```

Example “custom” reduction

```
class MyMin : ReduceScanOp { //finds min element (equiv. to built-in “min”)
    type eltType;                //type of elements
    var soFar : eltType = max(eltType); //minimum so far

    proc accumulate(val : eltType) {
        if(val < soFar) { soFar = val; }
    }

    proc combine(other : MyMin) {
        if(other.soFar < soFar) { soFar = other.soFar; }
    }

    proc generate() { return soFar; }
}

var theMin = MyMin reduce A;
```


What about scans?

- Instead of just getting overall value, also compute value for every prefix

A	2	1	4	3	1	3	0	2
sum	2	3	7	10	11	14	14	16

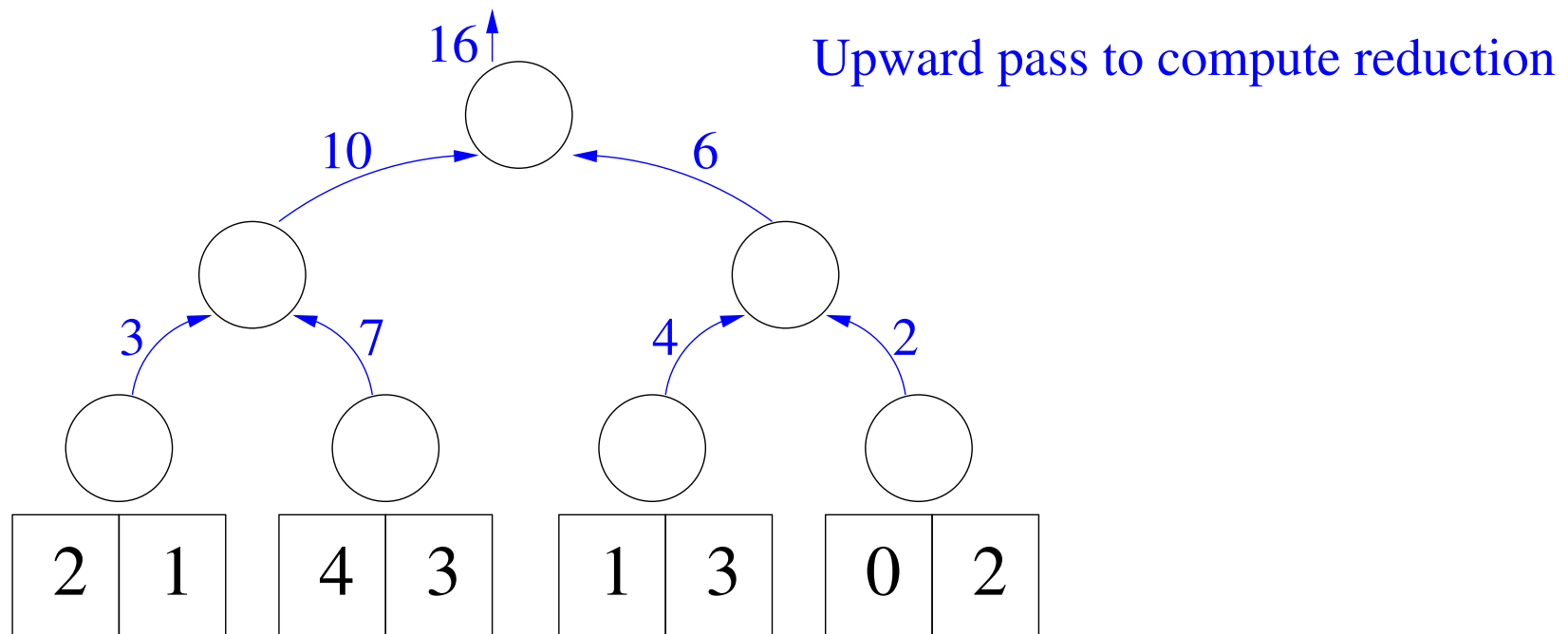
What about scans?

- Instead of just getting overall value, also compute value for every prefix

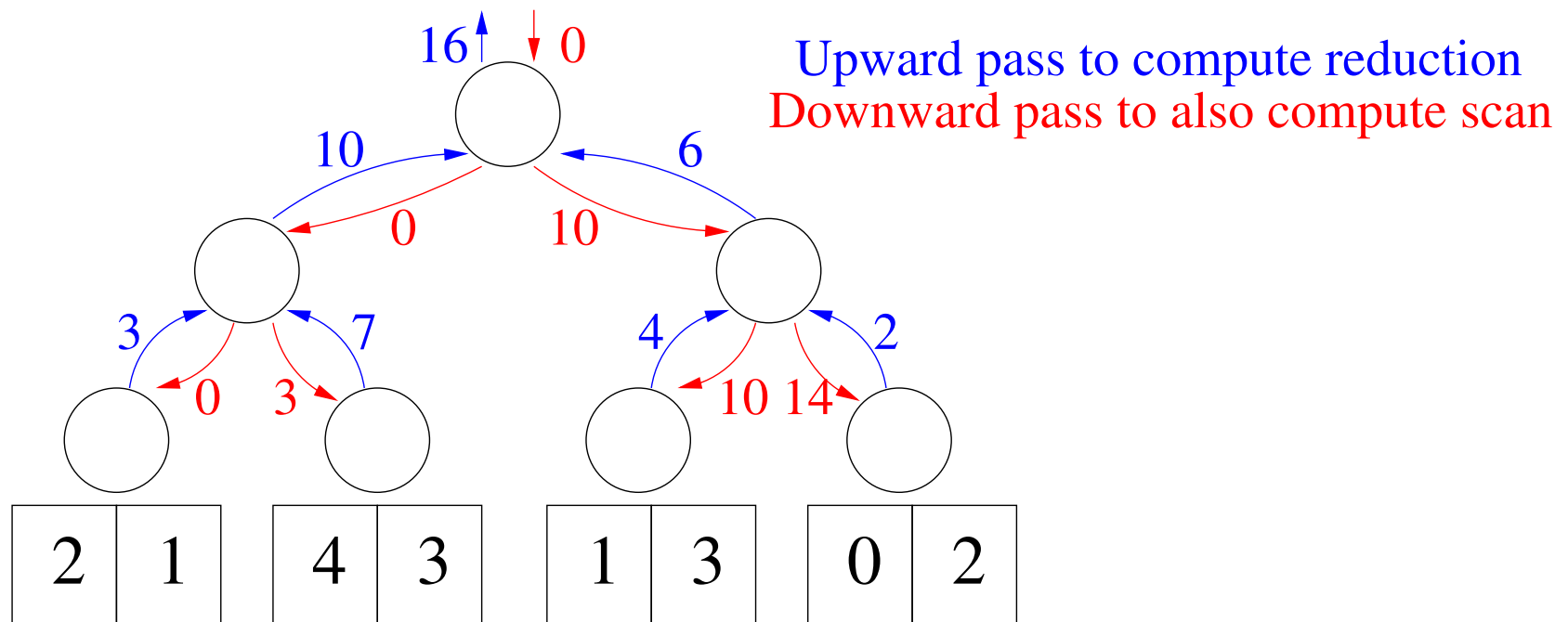
A	2	1	4	3	1	3	0	2
sum	2	3	7	10	11	14	14	16

`var minsArray = MyMin scan A;`

Computing the scan in parallel



Computing the scan in parallel



Presenting reductions

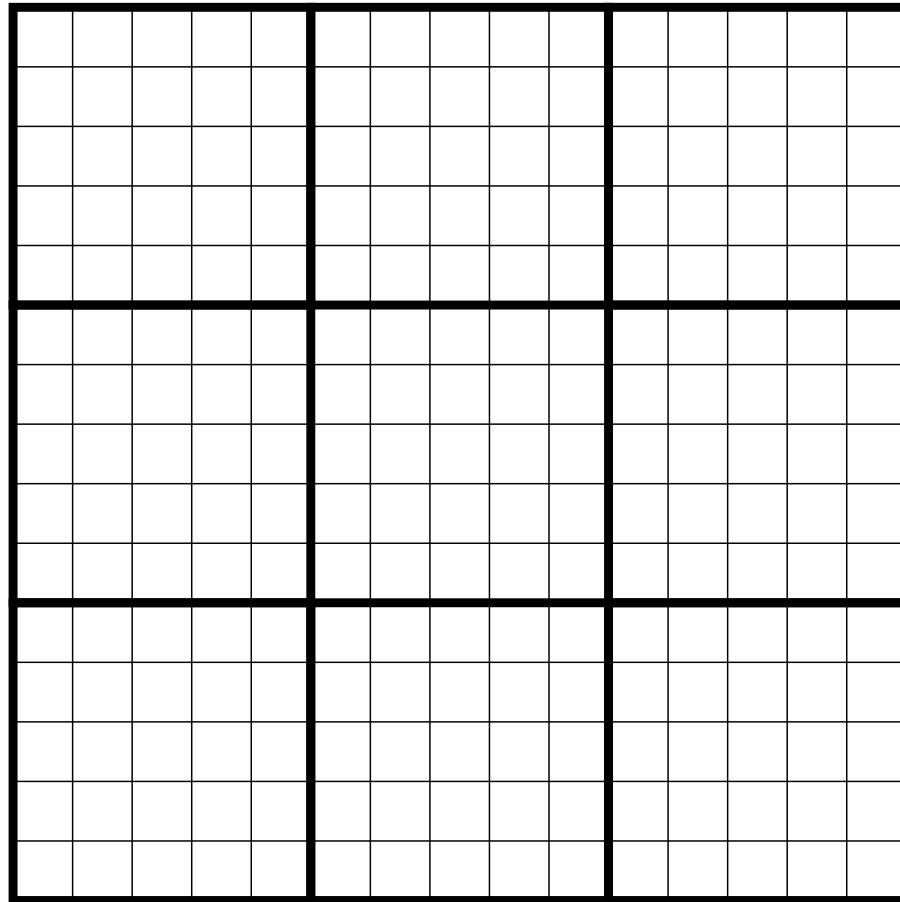
- Using reductions with standard functions
 - Optionally including scans
- Defining your own reductions

Parallel programming course

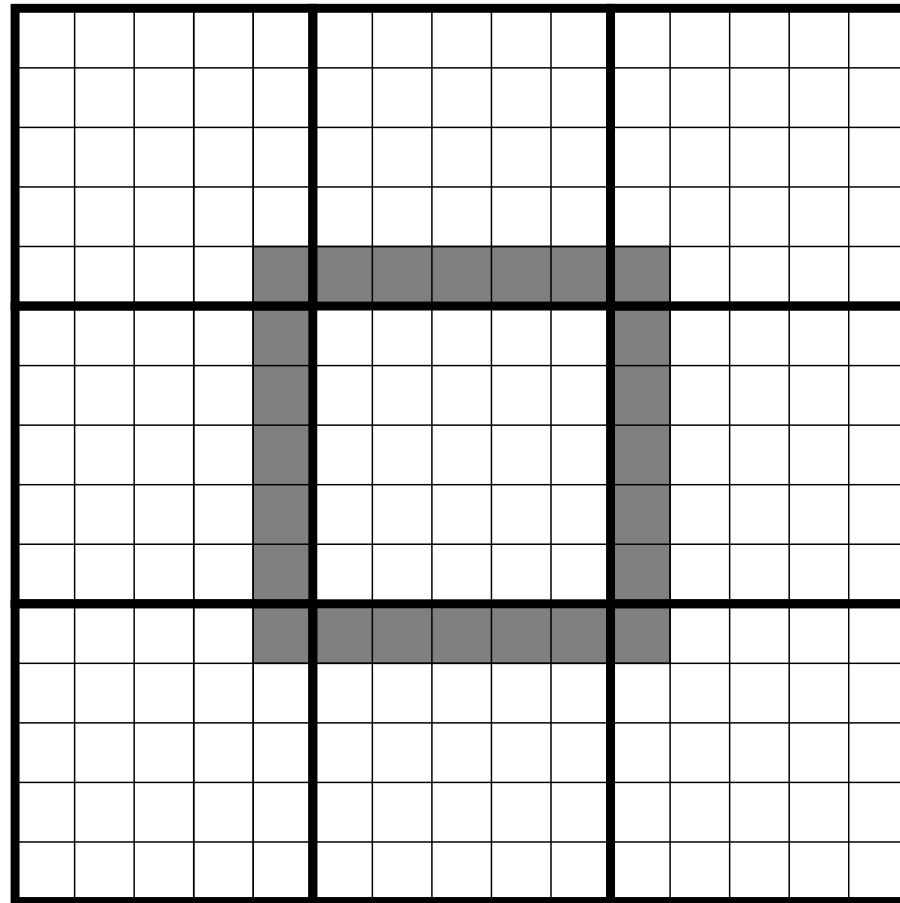
My experience

- Course to explore HPC overall
(apps, machines, system software, programming)
- Talked about Chapel (and ZPL) in contrast to MPI

Game of Life in MPI



Game of Life in MPI



Global-view

- Specify entire computation rather than one node's (local) view of it

```
var adjacentDomain : domain(2) = {x-1..x+1, y-1..y+1};  
var neighborDomain = adjacentDomain[currentBoard.domain];
```

```
var neighborSum = + reduce currentBoard[neighborDomain];  
neighborSum = neighborSum - currentBoard[x, y];
```

Representing locality

- Give control over where code is executed:
 on Locales[0] do
 something();
- and where data is placed:
 on Locales[1] {
 var x : int;
 }

Representing locality

- Give control over where code is executed:
 on Locales[0] do
 something();
- and where data is placed:
 on Locales[1] {
 var x : int;
 }
- Can move computation to data:
 on x do something();

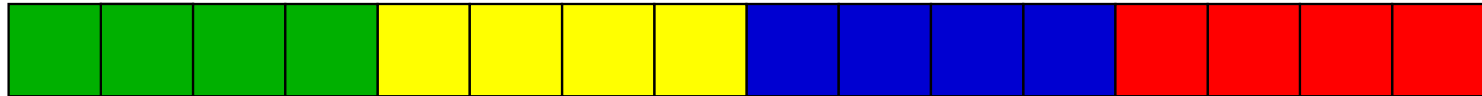
Separate from parallelism

- Serial but multi-locale:
 on Locales[0] do function1();
 on Locales[1] do function2();
- Parallel *and* multi-locale:
 cobegin {
 on Locales[0] do function1();
 on Locales[1] do function2();
 }

Managing data distribution

- Domain maps say how arrays are mapped

var A : [D] int dmapped Block(boundingBox=D)



var A : [D] int dmapped Cyclic(startIdx=1)



Useful references

- B.L. Chamberlain, S.-E. Choi, E.C. Lewis, C. Lin, L. Snyder and W.D. Weathersby. "The case for high level parallel programming in ZPL". IEEE Computational Science and Engineering 5(3): 76-86, 1998. [link](#)
- Lots of stuff on Chapel website
 - H. Burkhart, M. Sathe, M. Christen, O. Schenk, and M. Rietmann. "Run, Stencil, Run! HPC Productivity Studies in the Classroom". Proc. 6th Conf. Partitioned Global Address Space Programming Models (PGAS), 2012. [link](#)

Take home: Parallel course

- Can demonstrate standard concepts
- Particularly suited to demonstrate global-view and locality management
- Lots of possible reading material to expose research element

Hands-on time

<http://cs.colby.edu/kgburke/?resource=sigcse>

Summary / discussion

How else might you use Chapel?

- Operating Systems
 - Easy thread generation for scheduling projects
- Software Design
 - Some parallel design patterns have lightweight Chapel implementations
- Artificial Intelligence
 - (or other courses w/ computationally-intense projects)
- Independent Projects

Caveats

- Still in development
 - Error messages thin
 - New versions every 6 months
 - Not many libraries
 - (Students thought this was awesome!)
- No development environment
 - Command-line compilation in Linux

Conclusions

- Chapel is easy to pick up
- Chapel can be used in many courses
- Loads of features, but...
- Flexible depth of material
- Students will dig in!

Your Feedback

- What are your impressions of Chapel?
- How likely are you to adopt Chapel?
 - What course(s) will you use it in?
- What resources would help you adopt it?
 - Kyle has a bunch and is happy to share!!!

Thanks!

dbunde@knox.edu

kgburke@colby.edu