Using Chapel to teach parallel concepts

David Bunde
Knox College
dbunde@knox.edu

Acknowledgements

- Silent partner: Kyle Burke
- 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

- 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

Outline

- Introduction to Chapel
  - Why Chapel?
  - Basic syntax
  - Parallel keywords
  - Reductions
  - Features for distributed memory
- Using Chapel in your courses
- Hands-on time

Your presenter is...

- Interested in high-level parallel programming
- Enthusiastic about Chapel and its use in education

- **NOT connected to Chapel development team**

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

Chapel Resources

- Materials for this workshop

  http://faculty.knox.edu/dbunde/teaching/chapel/CCSC-CP14/

- 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)

Basic syntax

“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

Parallel keywords

Parallel Loops

- Two kinds of parallel loops:

  forall i in 1..10 do statement; //omit do w/ braces

  coforall i in 1..10 do statement;

- forall creates 1 task per processing unit
- coforall creates 1 per loop iteration
  - Used when each iteration requires lots of work and/or they must be done concurrently

Asynchronous Tasks

- Easy asynchronous task creation:

```
begin statement;
```

- Easy fork-join parallelism:

```
cobegin {
  statement1;
  statement2;
  ...
}     //creates task per statement and waits here
```

Sync blocks

- sync blocks wait for tasks created inside it
- These are equivalent:

```
sync {                          cobegin {
  begin statement1;               statement1;
  begin statement2;               statement2;
  ...                             ...
}                               }
```

Sync variables

- sync variables have value and empty/full state
  - store ≤ 1 value and block operations can't proceed

- Can be used as lock:

```
var lock : sync int;
lock = 1;               //acquires lock
...
var temp = lock;        //releases the lock
```

Example for teaching parallelism

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

Task creation with begin

```
var lowTotal : int = 0;
var highTotal : int = 0;

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

var total = lowTotal + highTotal;
```

Task creation with begin

```
var lowTotal : int = 0;
var highTotal : int = 0;

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

var total = lowTotal + highTotal;
```

Incorrect: race condition

Correct implementation w/ begin

```
var lowTotal : int = 0;
var highTotal : int = 0;
sync {
    begin ref(lowTotal) {
        for i in 1..50 do lowTotal += i;
    }
    begin ref(highTotal) {
        for i in 51..100 do highTotal += i;
    }
}
var total = lowTotal + highTotal;
```

A common pattern

```
sync {
    begin task1();
    begin task2();
    begin task3();
}
```

cobegin: Syntactic sugar

```
cobegin {
    task1();
    task2();
    task3();
}
```

Forall loops

```
var total : int = 0;
forall i in 1..100 {
    total += i;
}
```

Forall loops

```
var total : int = 0;
forall i in 1..100 {
    total += i;
}
```

Why doesn't this work?

Fixing the race

```
var total : sync int = 0;
forall i in 1..100 {
    total += i;
}
```

More sugar: forall shortened

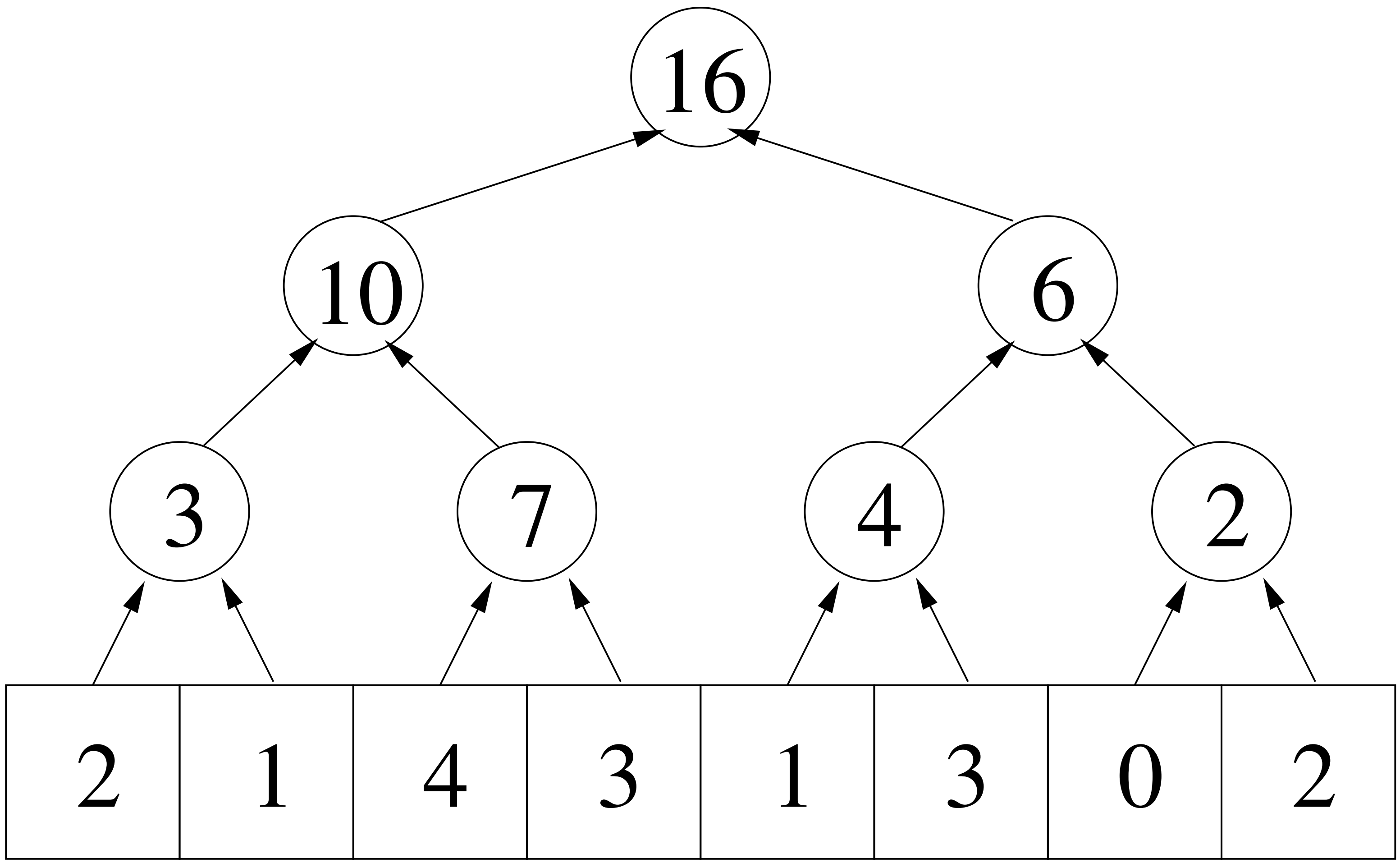
```
var total : sync int = 0;
forall i in 1..100 {
    total += i;
}
[i in 1..100] total += i;
```

Reductions

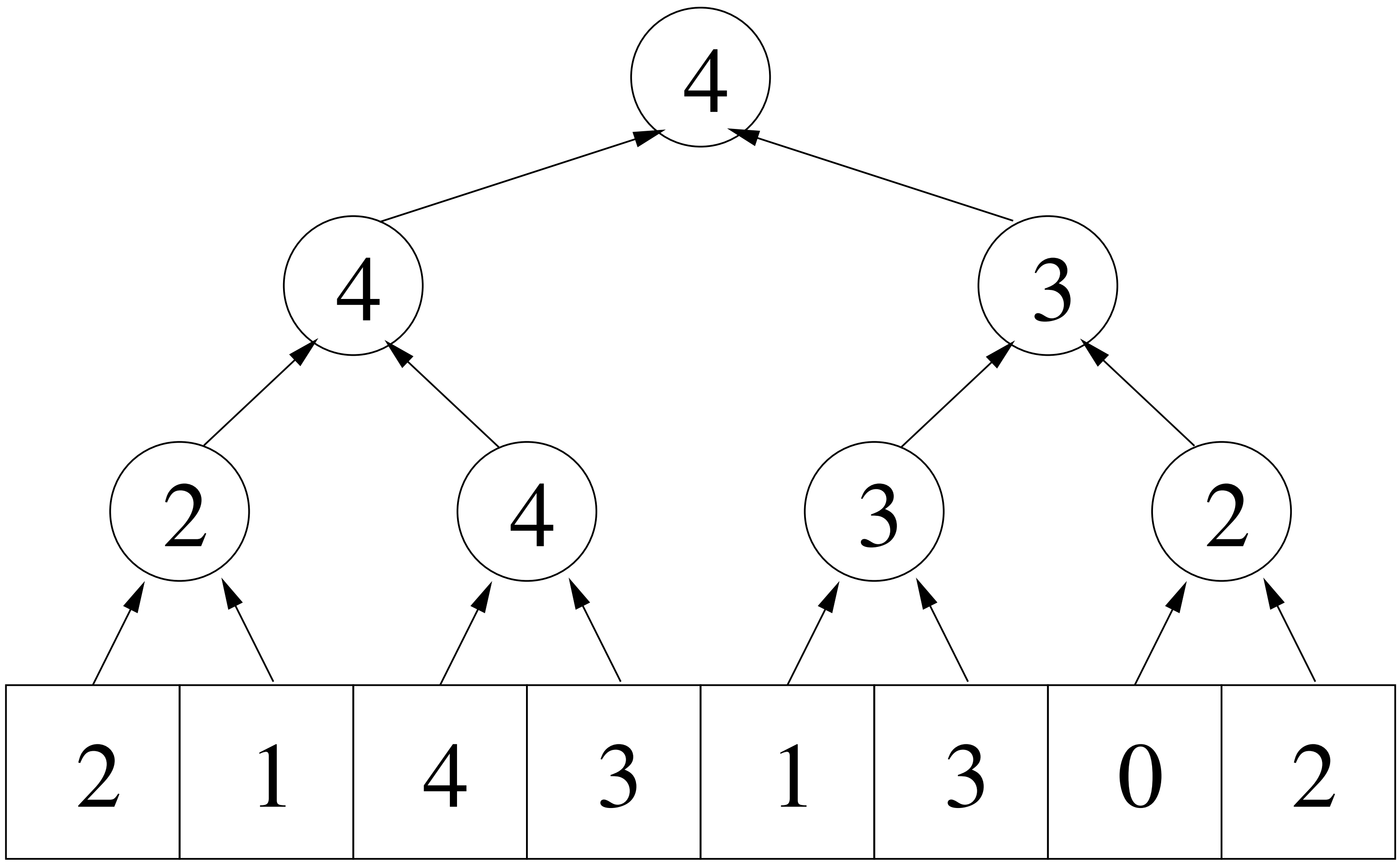
Summing values in an array

2	1	4	3	1	3	0	2

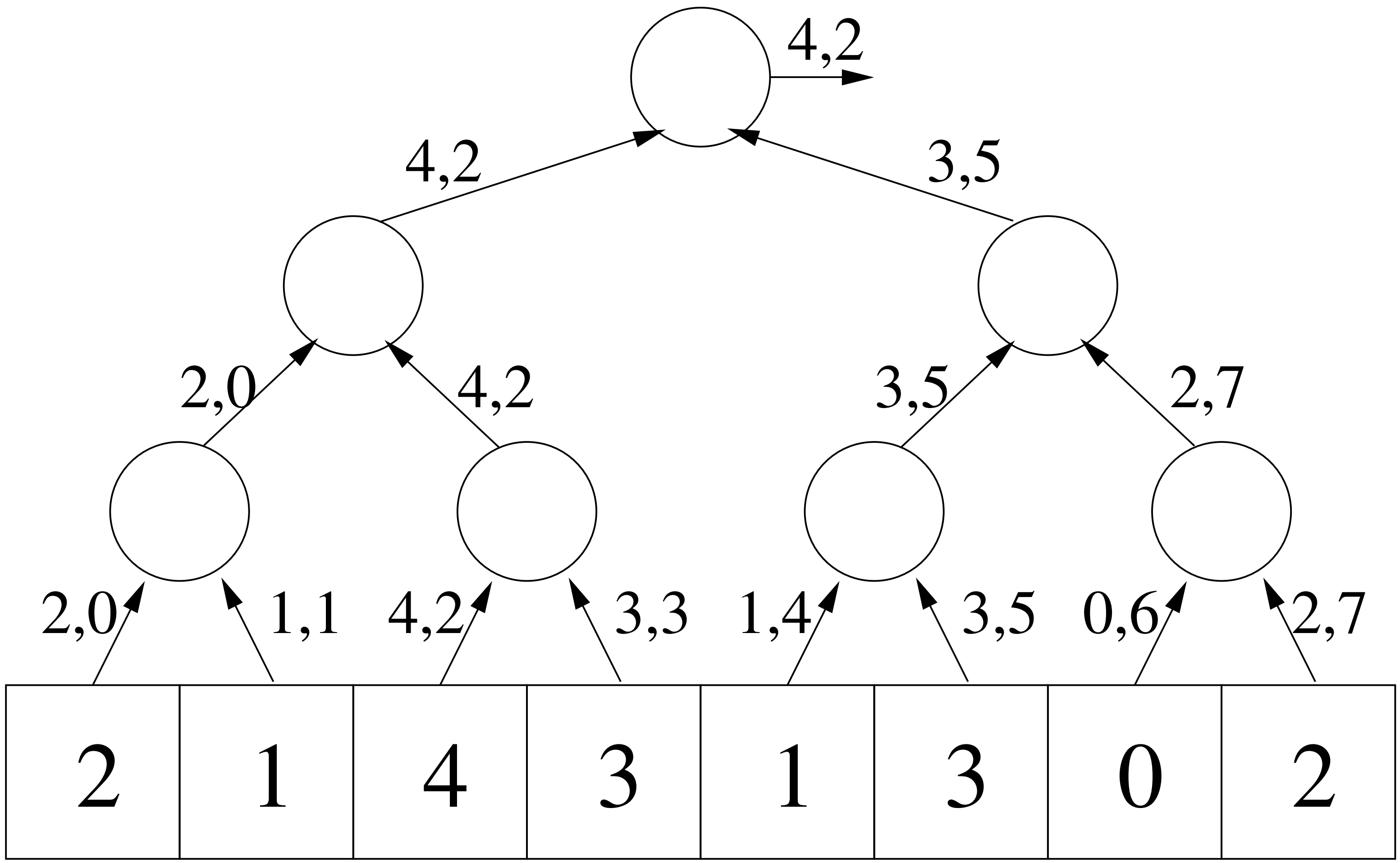
Summing values in an array



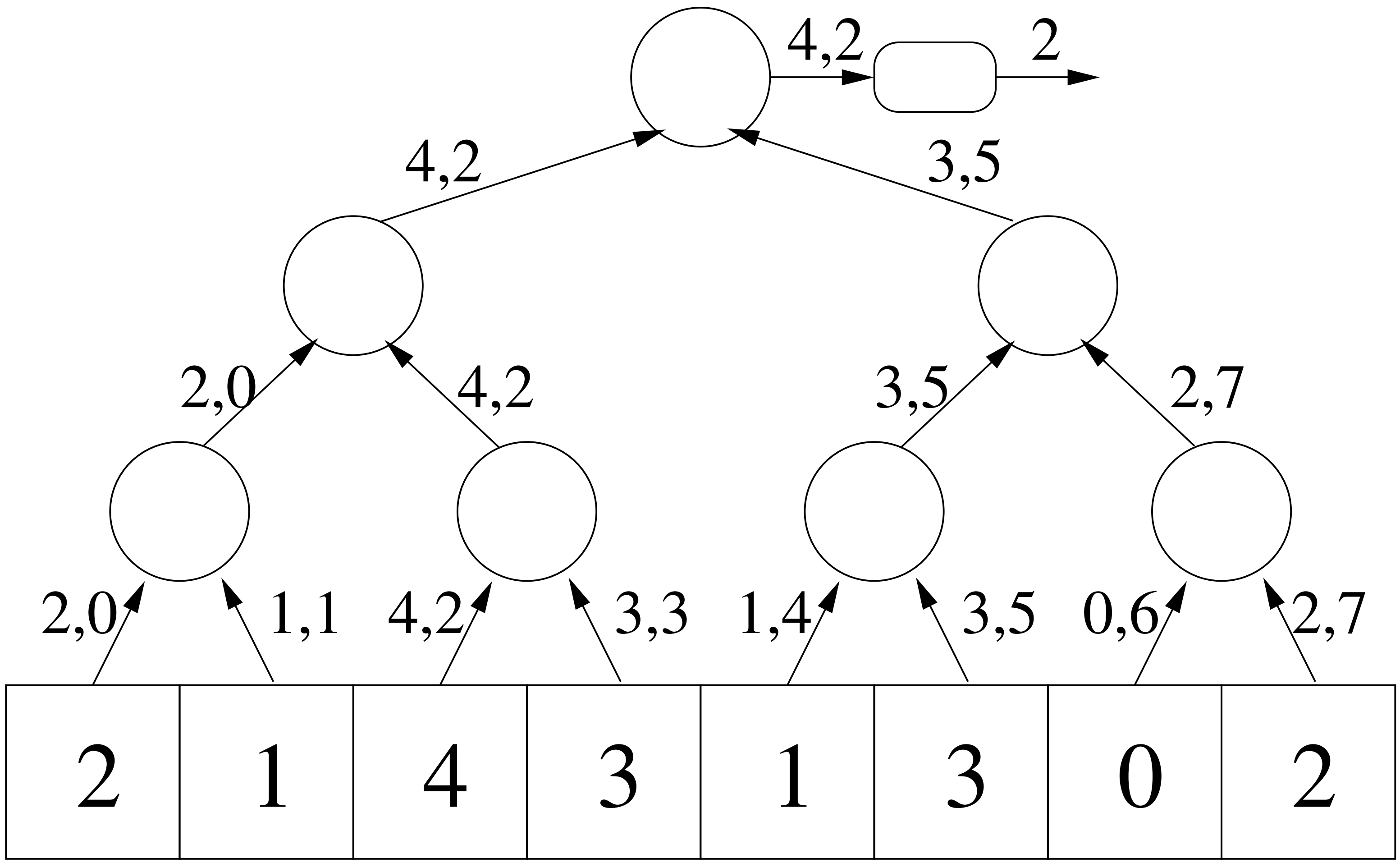
Finding max of an array



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 index

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 index
- Init: Create "empty" tally
- Accumulate: Add single value to 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 index
- Init: Create "empty" tally

  return (MIN_INT, -1)
- Accumulate: Add single value to tally

  (larger value, its index)

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, 2^{nd} largest

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 π/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
```

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;
```

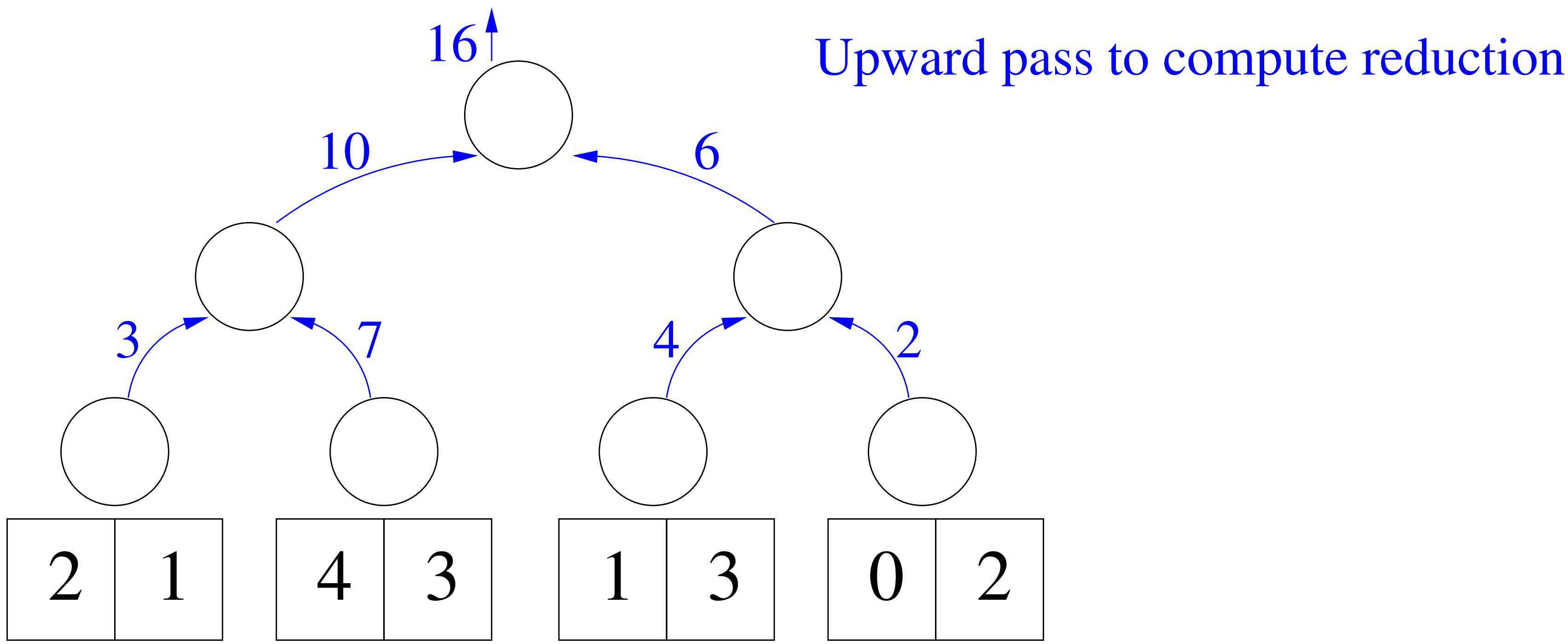
Scans

- 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 sum = + scan A;
```

Computing the scan in parallel



Representing locality

- Give control over where code is executed:

```
on Locales[0] do
    something();
```

Features for distributed memory

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();
```

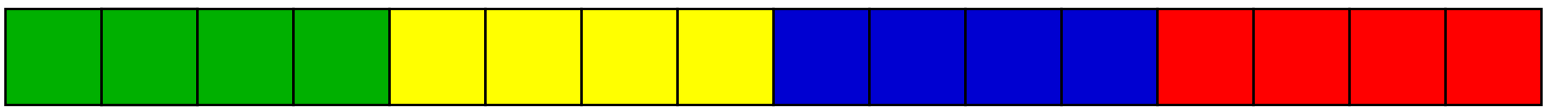
- Parallel *and* multi-locale:

```
cobegin {
    on Locales[0] do function1();
    on Locales[1] do function2();
}
```

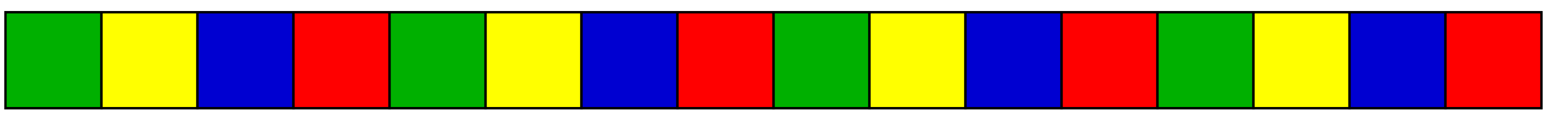
Managing data distribution

- Domain maps say how arrays are mapped

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



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



Using Chapel in your courses

Parallel programming

- Can demonstrate standard concepts
- “Global view” language
- Particularly suited to demonstrate
  - reductions (and scans)
  - data layout and locality management
- Topic of many research papers for a seminar

Programming languages

- “Parallel paradigm”
- Can illustrate both task and data parallelism
- “Global view”
- Expressively represent locality
- Data layout is orthogonal to operations on it
- Language evolving and design is actively being discussed

Algorithms

- Introduce only basic Chapel
  - Assign tutorial and give minimal introduction

- Parallel divide & conquer and/or brute force

- Reductions and custom reductions

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 with computationally-intense projects)
- Independent Projects

Caveats

- Still in development
  - Error messages thin
  - New versions every 6 months
  - Not many libraries
- No development environment
  - Command-line compilation in Linux

Conclusions

- Chapel is easy to pick up
- Flexible depth of material
- Suitable for many courses
- Still plenty to do for teaching it

Conclusions

- Chapel is easy to pick up
- Flexible depth of material
- Suitable for many courses
- Still plenty to do for teaching it

Let me know how you use it!

dbunde@knox.edu

Hands on time

(and/or break)

http://faculty.knox.edu/dbunde/teaching/chapel/CCSC-CP14/exercises.html