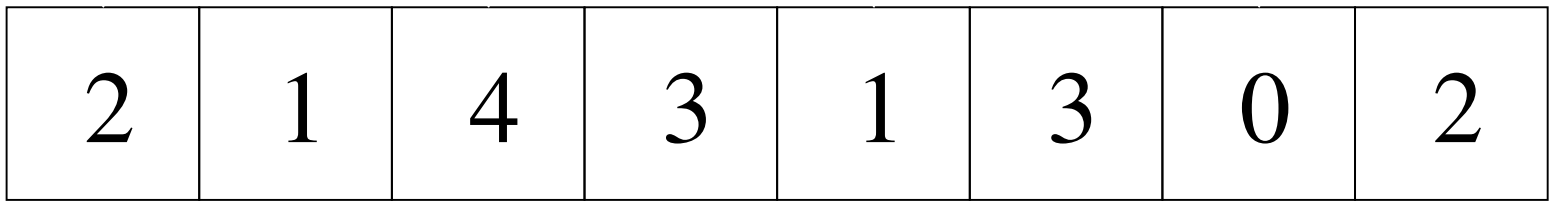
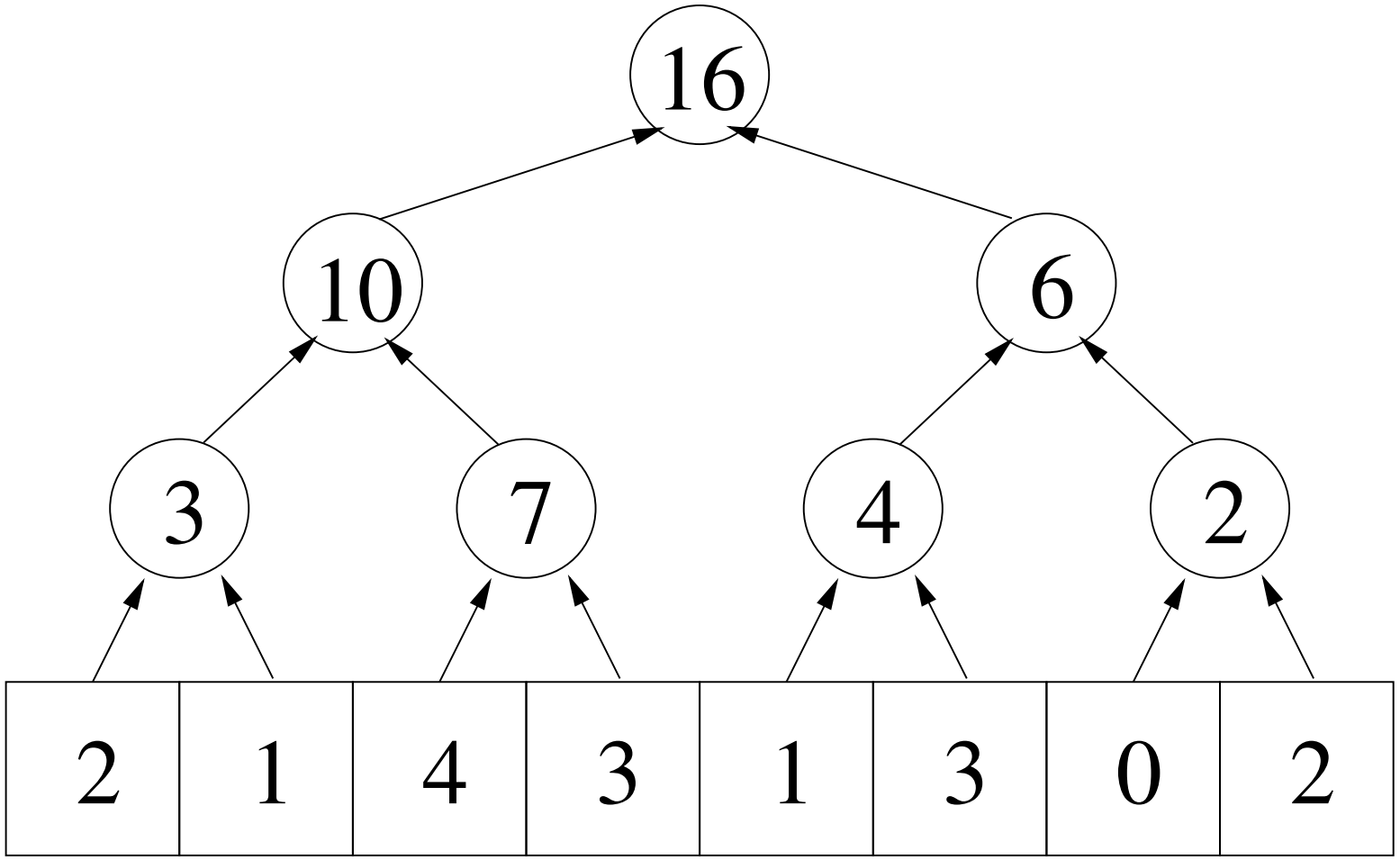
Reductions II: The Revenge

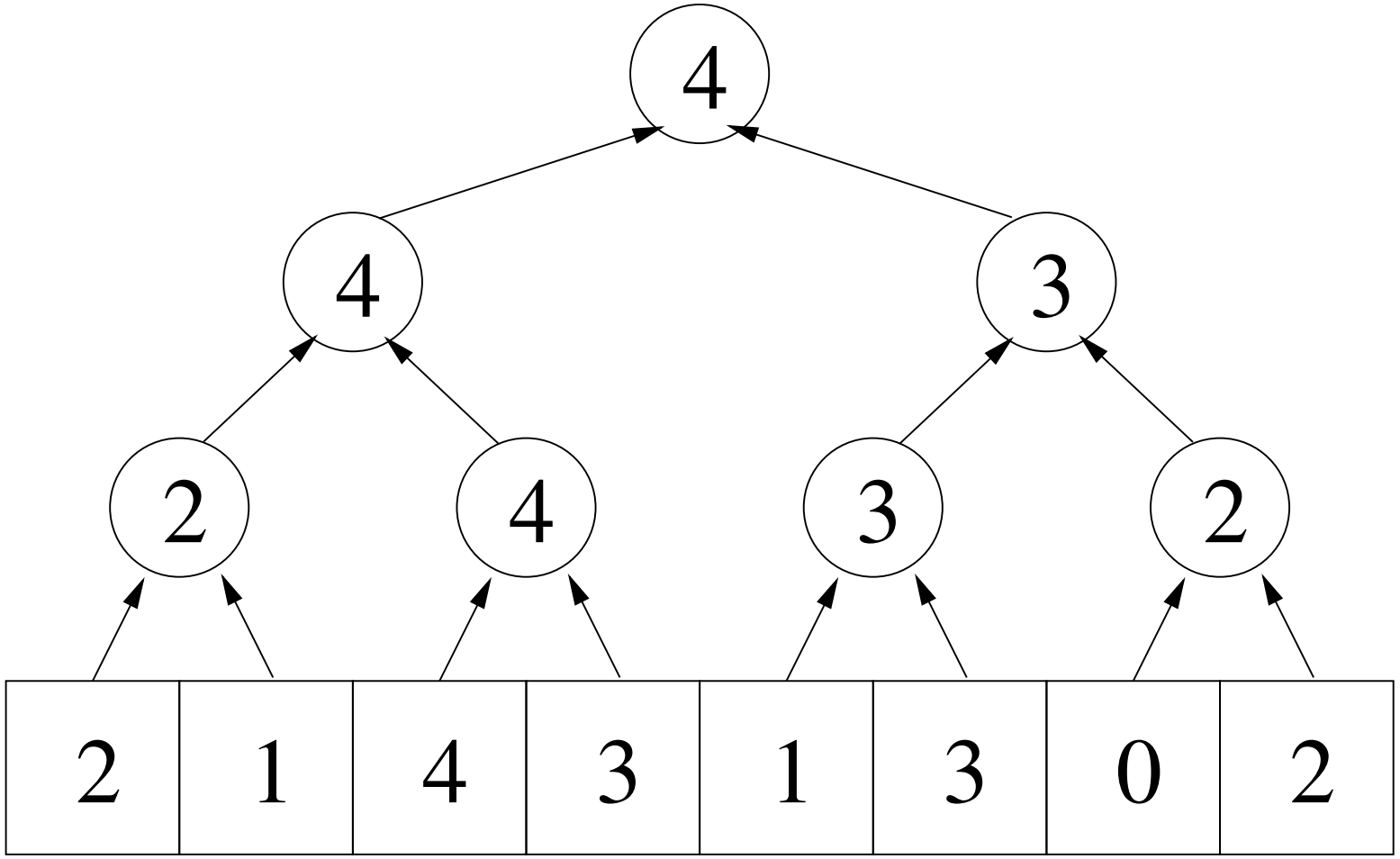
Summing values in an array



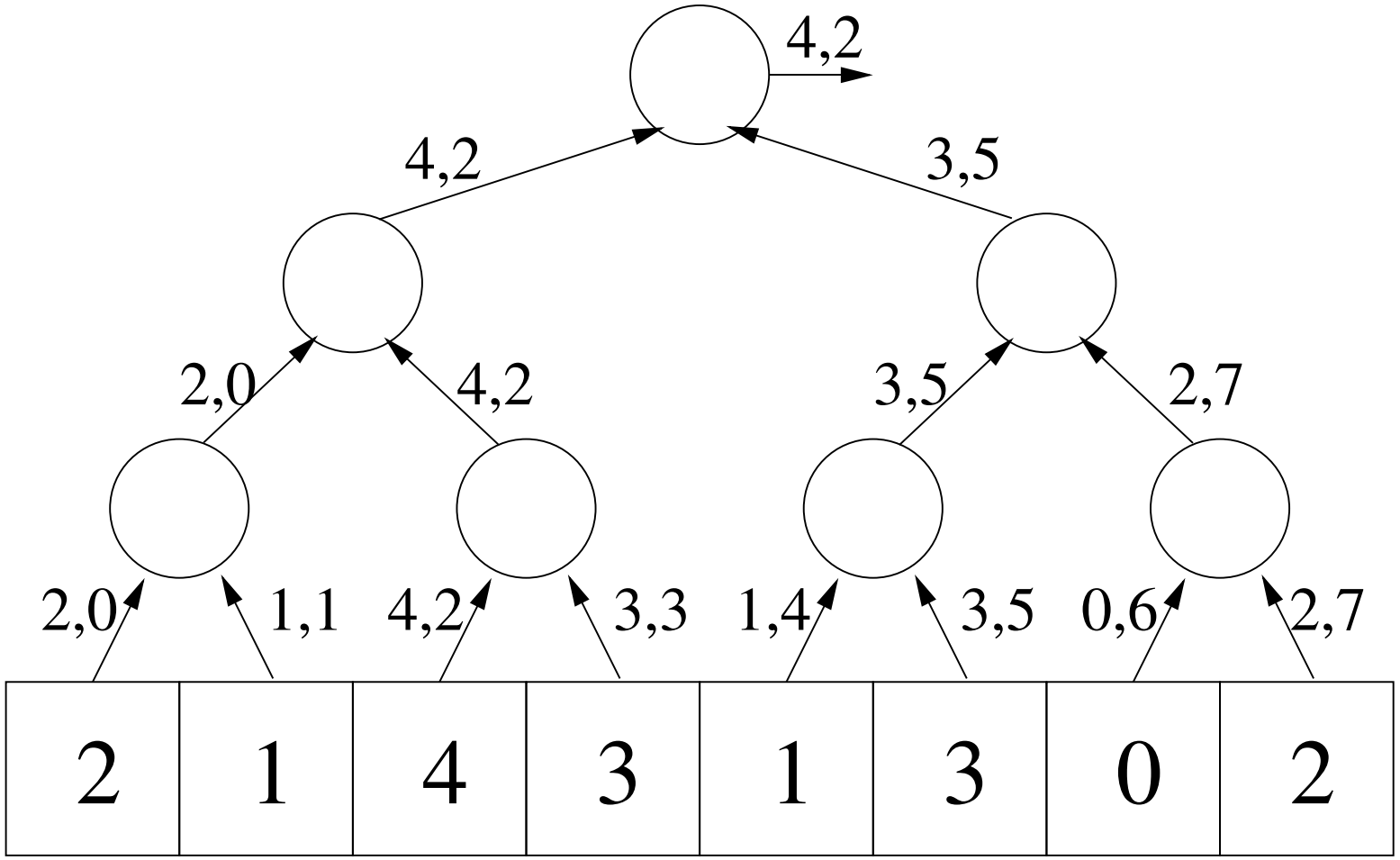
Summing values in an array



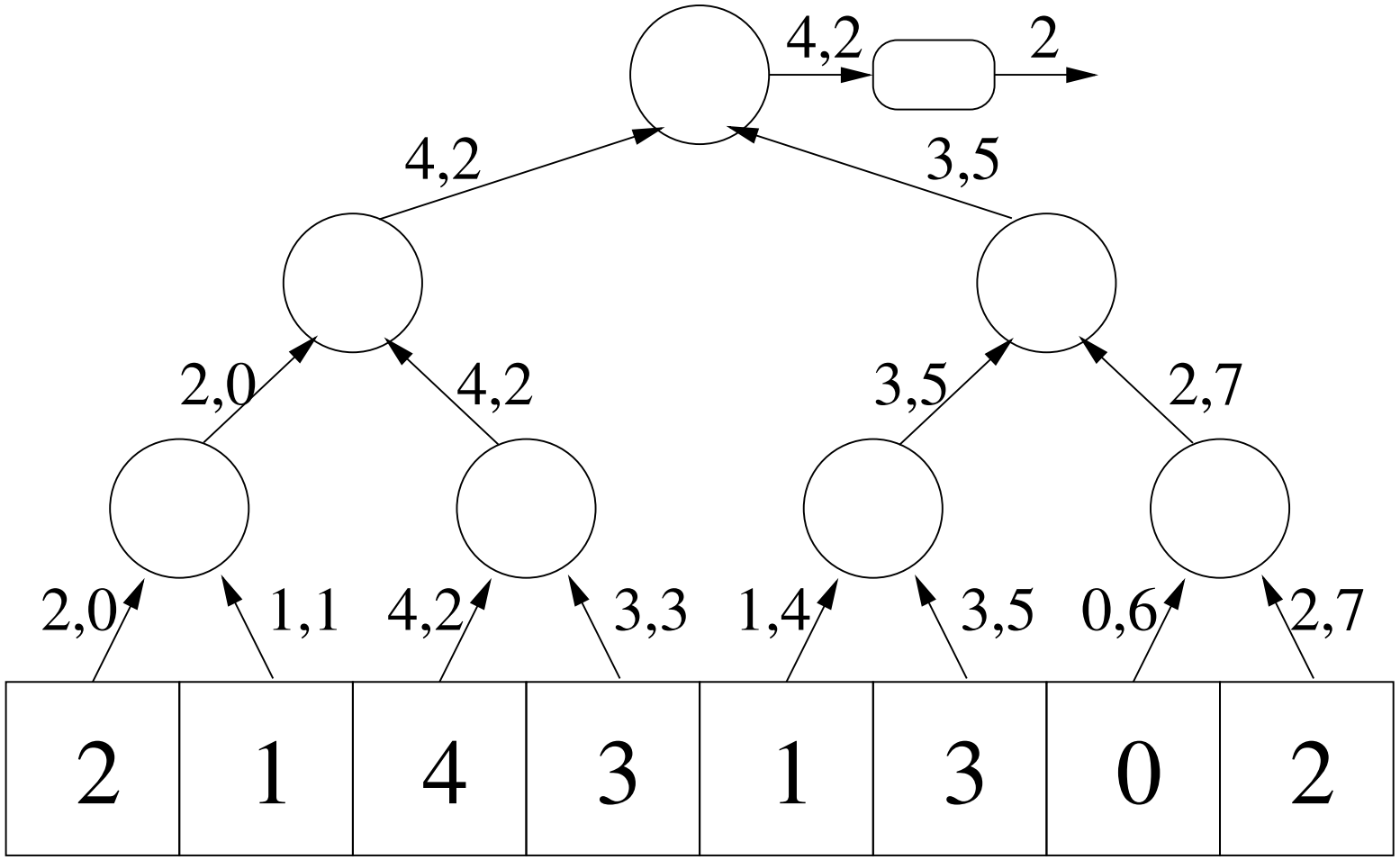
Finding max of an array



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

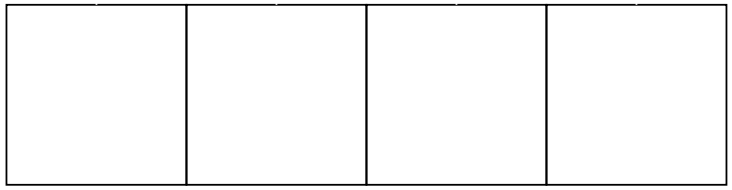
  (value, index)
- Combine: Combine 2 tallies

  take whichever pair has larger value
- Reduce-gen: Generate result from tally

  return the index

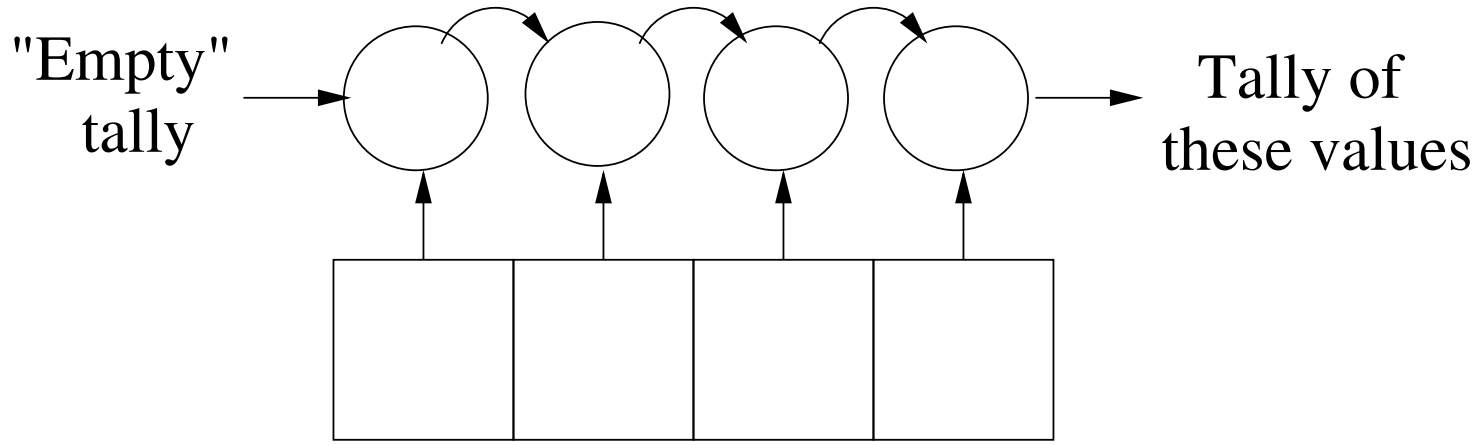
Two issues

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



Two issues

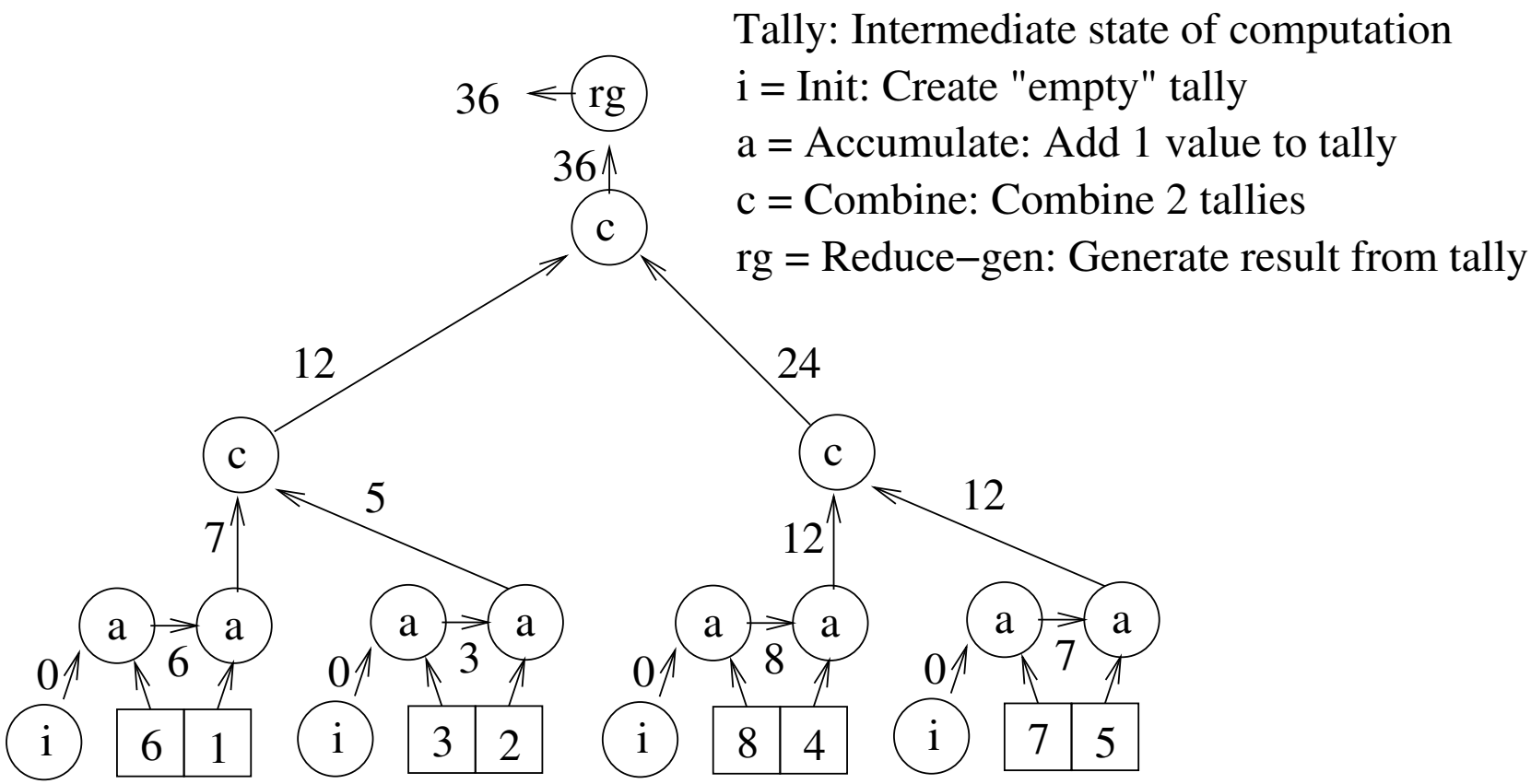
- Need to convert initial values into tallies
- May want separate operation for values local to a single processor



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 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
```

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

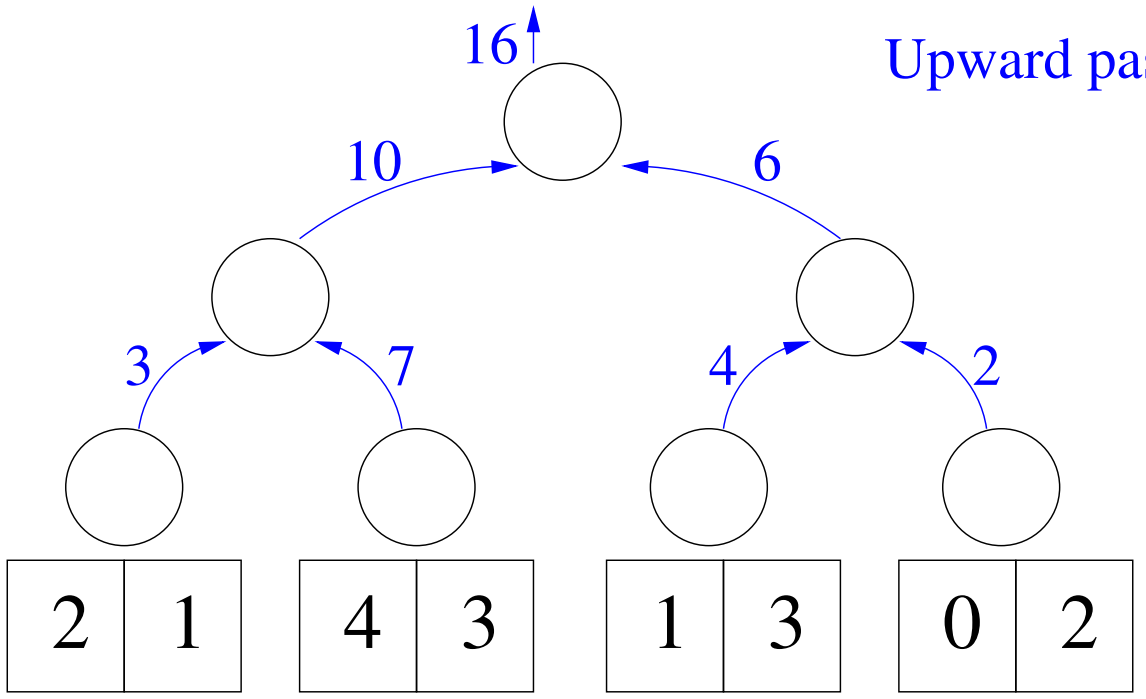
What about scans?

- 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

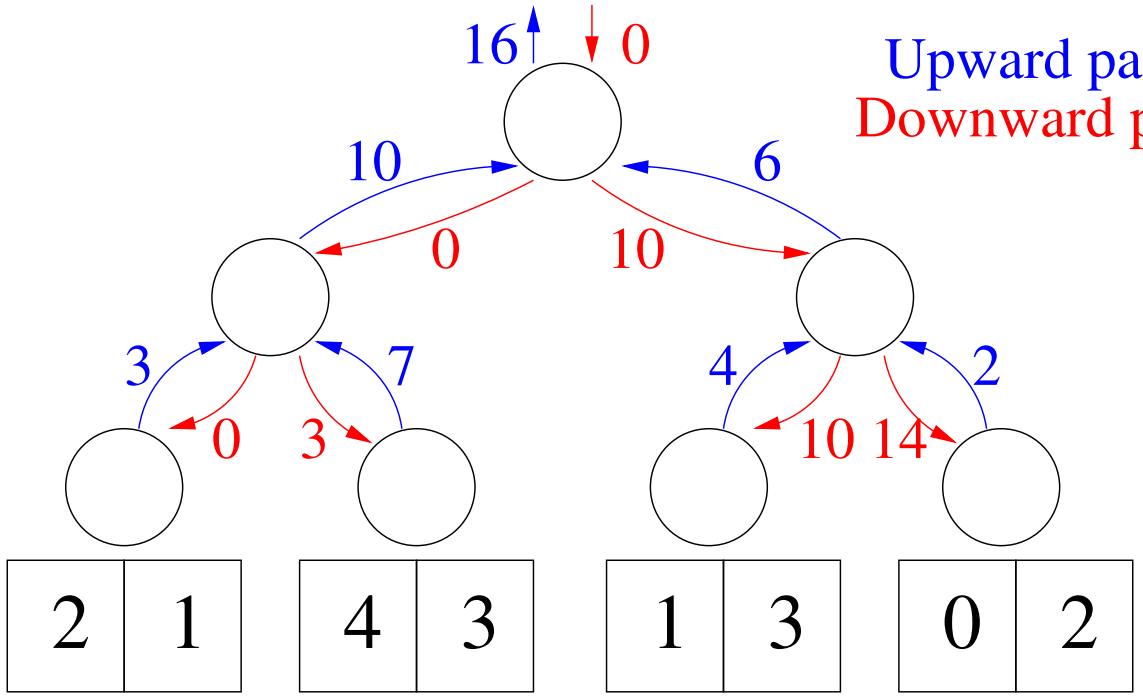
var minsArray = MyMin scan A;

Computing the scan in parallel



Upward pass to compute reduction

Computing the scan in parallel



Upward pass to compute reduction
Downward pass to also compute scan

Presenting reductions

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

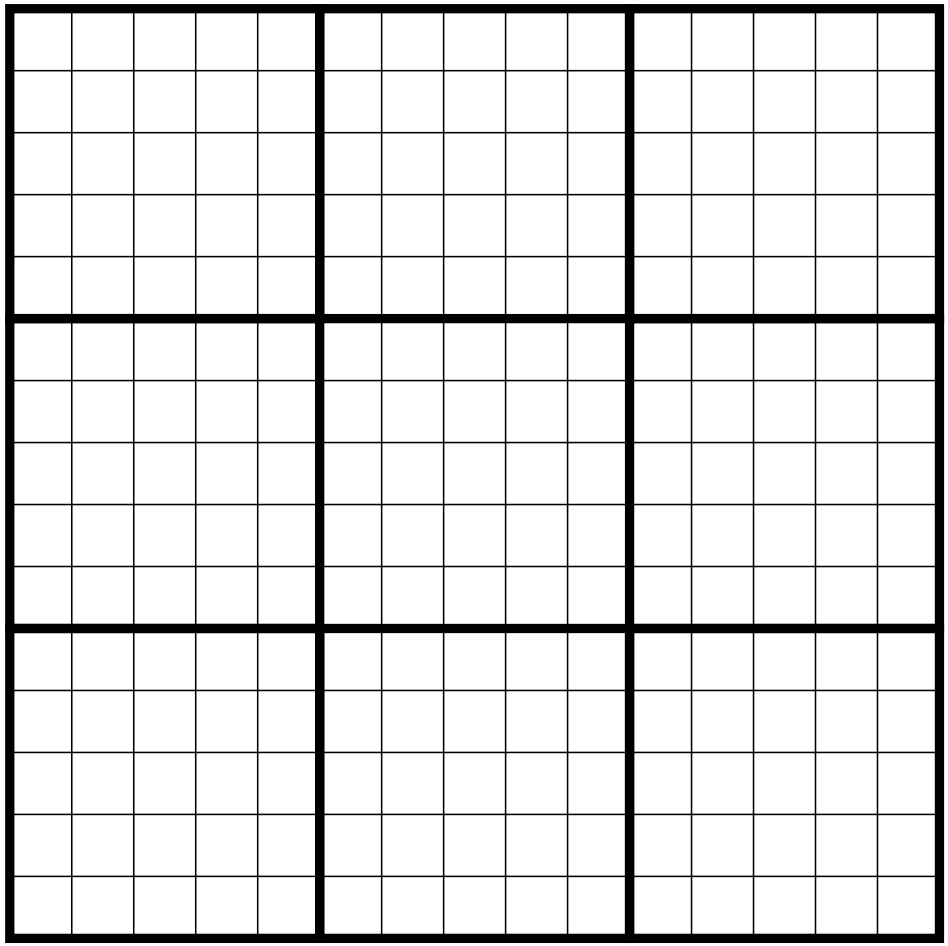
Parallel programming course

My experience

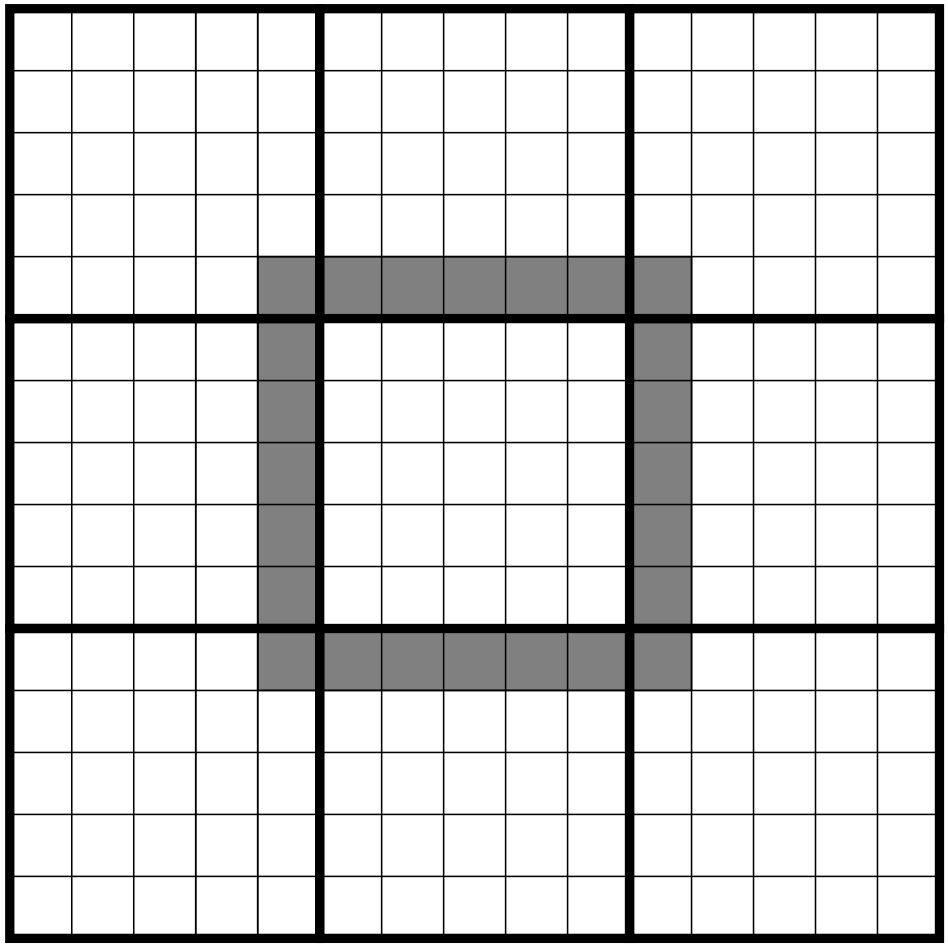
- Course to explore HPC overall

  (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();
```

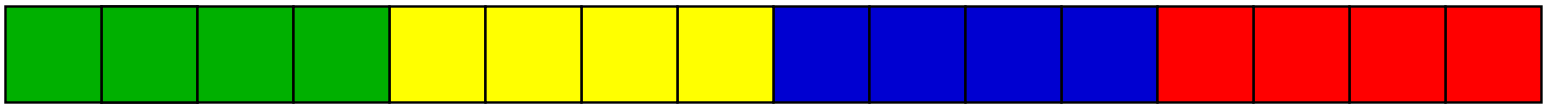
- Parallel *and* multi-locale:

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

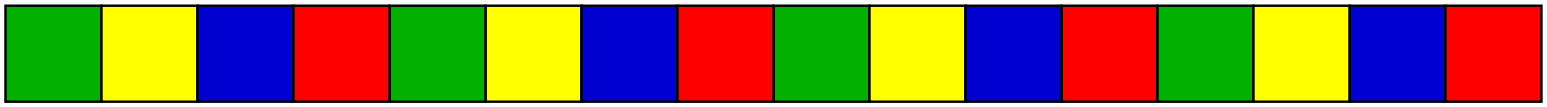
Managing data distribution

- 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