

High-level parallel programming using Chapel

David Bunde, Knox College

Kyle Burke, Wittenberg University

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 the SC Educator program, the Ohio Supercomputing Center, and 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



Schedule

- Part I: 1:30-3:00
 - Introduction to Chapel and the Workshop
 - Survey of Chapel Syntax
 - Hands-on Session 1
- Part II: 3:30-5:00
 - Advanced Ranges and Domains
 - Hands-on Session 2
 - Using Chapel in the Classroom

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; recent work on GPUs (see [Sidelnik et al., IPDPS 2012](#))
- Supports (but doesn't require) global-view programming, in which programmers express whole operation rather than specifying each processor's role

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: $x = + \text{ reduce } A$ //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

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

$B = + \text{ scan } A;$ //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/SC12/>
- Our tutorials
<http://faculty.knox.edu/dbunde/teaching/chapel/>
<http://www4.wittenberg.edu/academics/mathcomp/kburke/chapelTutorial.html>
- Chapel website (tutorials, papers, language specification)
<http://chapel.cray.com>
- Mailing lists (on SourceForge)

Accessing Practice Systems (during SC only)

- We have practice accounts set up for use during the workshop
- Get handout from one of the instructors

Installing Chapel Yourself

- Instructions (<http://chapel.cray.com/download.html>)
 - Download: <http://sourceforge.net/projects/chapel>
 - Unzip file
 - Enter chapel-1.6 directory and invoke make
 - source util/setchplenv.csh or util/setchplenv.sh to set environment variables
- For multiuser installations (e.g. in /usr/local):
<http://faculty.knox.edu/dbunde/teaching/chapel/install.html>

Survey of Chapel 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 can contain the following:
var/const identifier : type = initial_value;
- var or const: variable or named constant
- Basic types are int, real, boolean, string
- Also supports imaginary and complex values:
var x : imag = 1.0i;
var y : complex = 1.2 + 3.4i;
- Type is optional if it can be inferred from initial value

Config Variables

- Optionally set from the command line; they're Chapel's alternative to command-line args
- Declared with config:

```
config var x = 0;    //0 unless overridden on
                    // command line
```
- Set on command line with two dashes: --

```
./hello --x=23      //runs hello with x set to 23
```

Operators

- Most operators are familiar: +, -, *, <, >, <=, ...
- = for assignment, == for equality testing
- / is integer division if both arguments are int
- Colon for casts:
 - var x = 3.14 : int; //casts to int (truncates)
 - var y = 2:real / 3; //promote 2 to 2.0 before division
- ** for exponentiation: 2**3 results in 2³
- <=> swaps value of two variables

Console I/O

- Output uses `write` and `writeln`, which support multiple arguments:

```
writeln("The value of x is ", x);
```

- Input uses `stdin.read` and `stdin.readln`:

```
stdin.read(x);
```

```
stdin.readln(x); //reads x and ignores rest of line
```

- Return value is boolean indicating whether something was read

Example: Reading until eof

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

Serial Control Structures

- if statements, while loops, and do-while loops are all pretty standard (we'll get to for loops)
- Difference: Statement bodies must either use braces or an extra keyword:
 - `if(x > 5) then y = 3;`
 - `while(x < 5) do x++;`
- Select is multi-way selection (switch in C/Java)

Procedures/Functions

```
proc name([arg_type] arg1 : type1, ...) : return_type {  
    body (with return statement(s))  
}
```

- Omit return_type for a function with no return value
(or if the type can be inferred)
- arg_type controls how arguments are passed:
 - const or omitted: variable is constant within function (exceptions on ref sheet)
 - in: pass by value (value copied into function)
 - ref: pass by reference
 - inout: value copied both in and out
 - out: final value copied back to calling block
- Omit argument types to write generic functions

Procedures/Functions (2)

- Can include default values for arguments by putting assignment in parameter list

```
proc f(x: int = 5) { ... }
```

- Can have a main procedure without arguments as program starting point

Ranges (Take 1)

- $i..j$ denotes the range containing $i, i+1, \dots, j$
- The endpoints can be variables
- Range is empty if 2nd value is less than 1st
- Can declare ranges as variables:

```
var R : range = 1..10;
```


Arrays

- Ranges can be used to declare arrays
- Indices determined by the range:

```
var A : [1..10] int; //declares A as array of 10 ints  
var B : [-3..3] int; //has indices -3 thru 3
```
- Array cells are accessed using indices:

```
A[1] = 23;  
A[2] = A[1] + 3;
```
- Arrays generate runtime out-of-bounds errors if invalid indices are used
- Can also create multi-dimensional arrays:

```
var C : [1..10, 1..10] int;
```

Domains

- Array creation actually requires a domain, which is the set of valid indices
- Anonymous domains created by putting range in brackets, but can also create domain variables:
var D : domain(1) = {1..10}; //domain of dimension 1
var A1 : [D] int;
var D2 : domain(2) = {1..10,1..10}; //domain of dim 2
var A2 : [D2] int;

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:

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

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

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 a domain, array, or anything supporting iteration

Parallel Loops

- To run loop iterations in parallel change for loop to forall or coforall:

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 in parallel

Asynchronous Tasks

- Can also create a specific task with begin:
begin statement; //create task for statement
- Can also create group of tasks and wait for all of them to finish (fork-join parallelism):
cobegin {
statement1;
statement2;
...
} //creates task for each statement and
//waits here for all to finish

Sync blocks

- sync blocks also wait for all tasks created within the block
- Example with equivalent cobegin block:

<code>sync {</code>	<code>cobegin {</code>
<code> begin statement1;</code>	<code> statement1;</code>
<code> begin statement2;</code>	<code> statement2;</code>
<code> ...</code>	<code> ...</code>
<code>}</code>	<code>}</code>

Sync variables

- sync variables have value and empty/full state
 - writing to an empty variable makes it full
 - reading from full variable makes it empty
 - attempt to write to a full variable blocks
 - reading from empty variable blocks

- Can be used to create a lock:

```
var lock : sync int;
```

```
lock = 1;           //acquires lock
```

```
...
```

```
var temp = lock;    //releases the lock
```


Reductions

- Express reduction operation in single line:
`var s = + reduce A; //A is array, s gets sum`
- Supports +, *, ^ (xor), &&, ||, max, min, ...
- Also minloc and maxloc, which return a tuple with min/max value and index where it occurs:
`var (val, loc) = minloc reduce A;`
- Can define custom reductions; need to define class to store partial work

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));
```

Scans

- Can also compute all partial results of a reduction using scan operation:

```
const R : range = 1..5;
```

```
const A : [R] int = [3, -1, 4, -2, 0];
```

```
var B : [R] int = + scan A; //B set to [3, 2, 6, 4, 4]
```

OO programming in Chapel

- Structures: Records and Classes
 - Several named variables combined into one object
 - Can have accompanying methods
 - Difference between them: Assignment copies contents of a record, but only a reference for a class

Circle as a Record

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

```
var c1, c2 : Circle;           //creates 2 Circle records  
c1 = new Circle(10);          /* uses system-supplied constructor  
                               to create a Circle record and  
                               copy its values into c1 */  
c2 = c1;                       //copies fields from c1 to c2
```

Circle as a Class

```
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 */
```

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

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; }
}
```

Timing your code

```
use Time; //includes Time module
```

```
var t : Timer; //declares a Timer variable
```

```
t.start();
```

```
//do whatever
```

```
t.stop();
```

```
writeln(t.elapsed(), " seconds elapsed");
```

Hands-on Session 1

[http://faculty.knox.edu/dbunde/teaching/chapel/
SC12/exercises.html](http://faculty.knox.edu/dbunde/teaching/chapel/SC12/exercises.html)

```
qsum -l -l nodes=1:ppn=12 -l  
walltime=1:30:00
```

Advanced Ranges and Domains

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

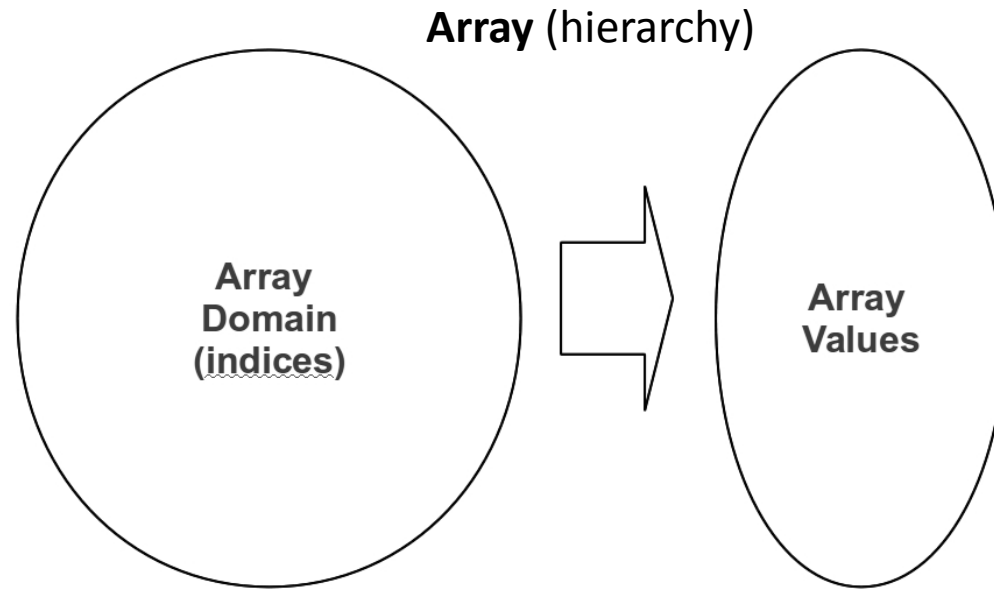
Chapel Domains

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

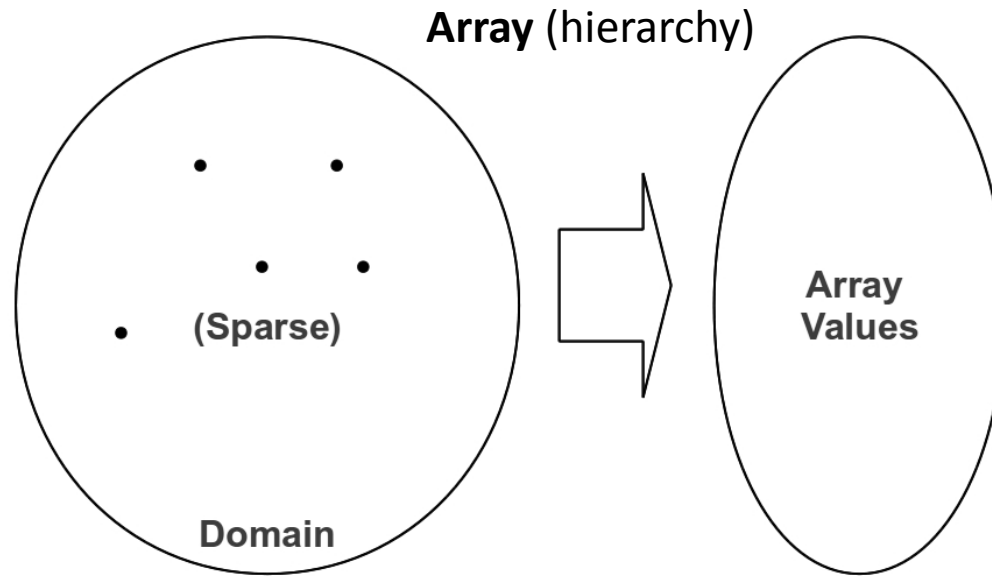
- Example:

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

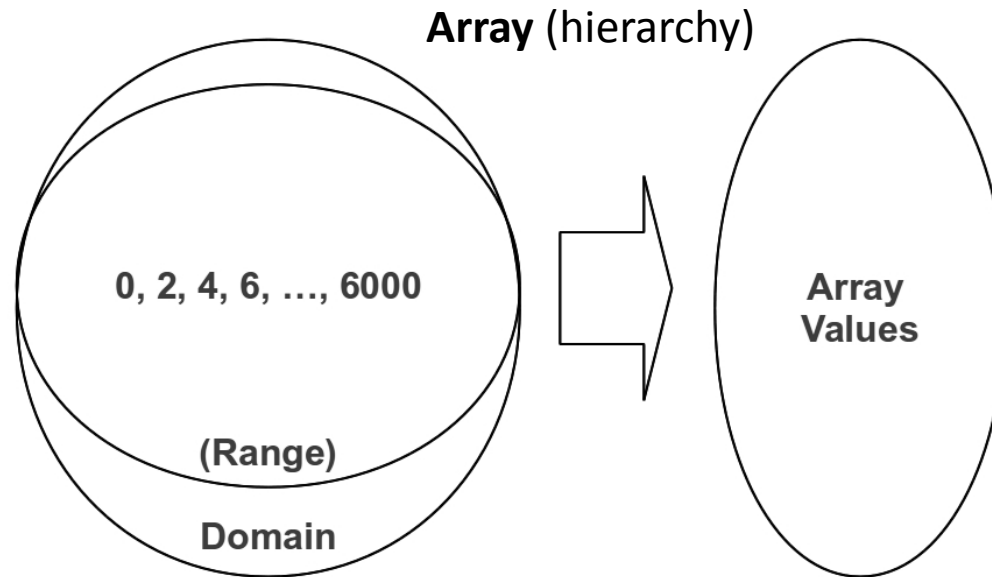
Chapel Domains



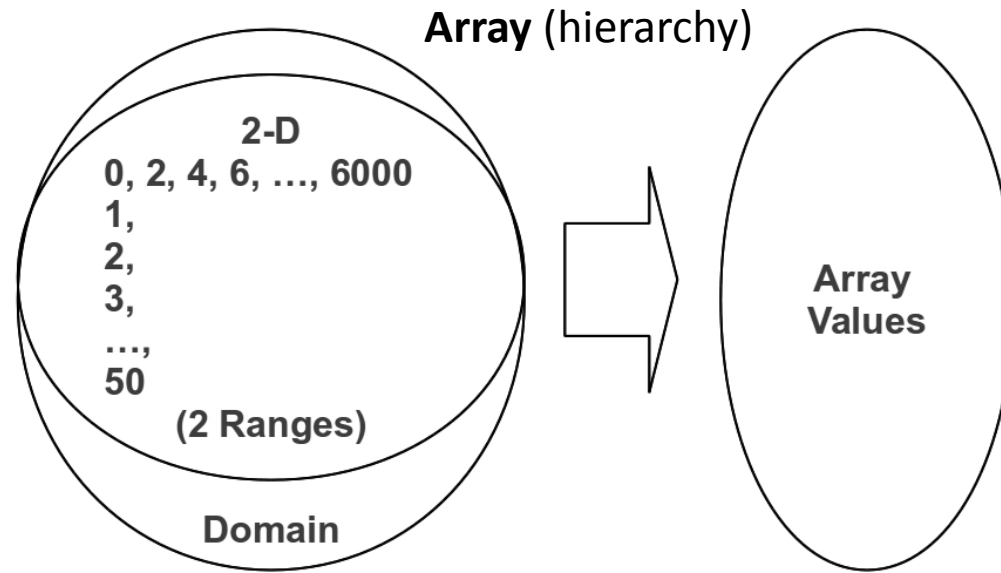
Chapel Domains



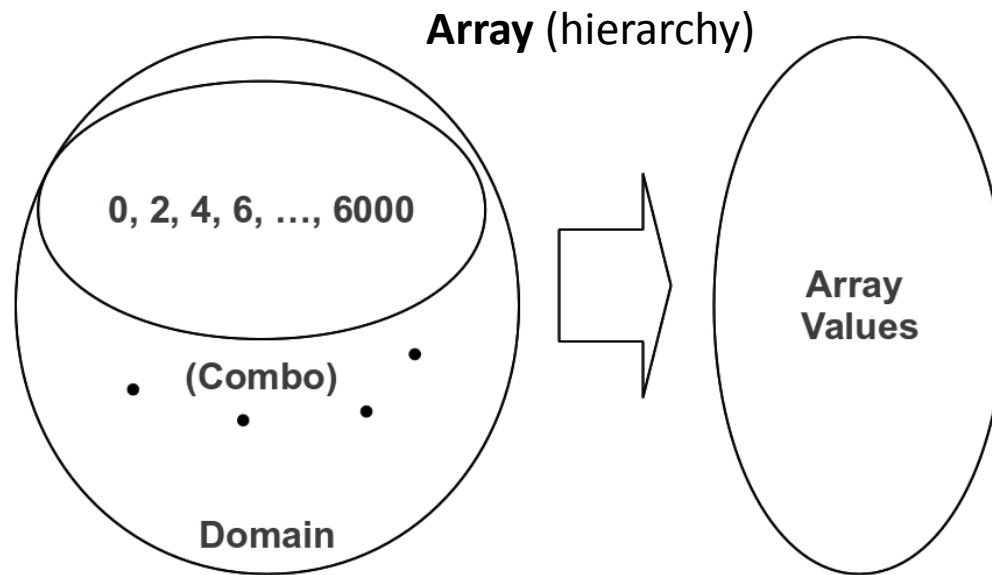
Chapel Domains



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

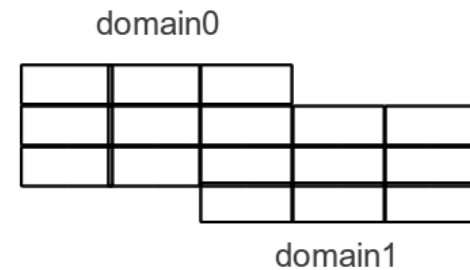


4	5	6	0
7	8	9	0

Domain Slices (Intersection)

domain0: [0..2, 1..3]

domain1: [1..3, 3..5]

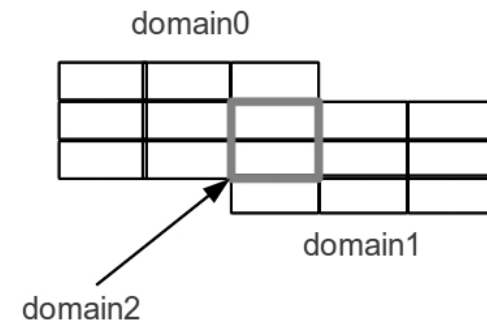


Domain Slices (Intersection)

domain0: [0..2, 1..3]

domain1: [1..3, 3..5]

domain2: [1..2, 3..3]



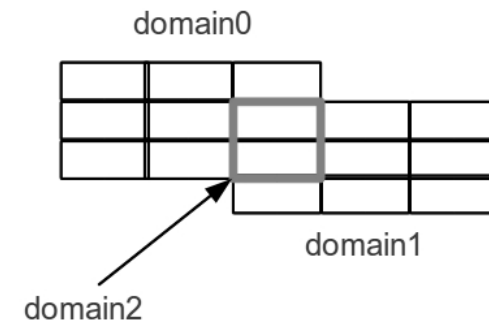
Domain Slices (Intersection)

//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];

Domains: Unbounded Game of Life

- Example of
 - Domain operations
 - One domain for multiple arrays
 - Changing domain for arrays
- Rules:
 - Each cell is either dead or alive
 - Adjacent to all 8 surrounding cells
 - Dead cell → Living if exactly 3 living neighbors
 - Living cell → Dead if not exactly 2 or 3 living neighbors

Unbounded? How?

- Plan: board starts with small living area, but can grow!
 - Start with 4x4 board

0	1	1	1
1	0	0	1
0	0	0	1
0	0	1	1

Unbounded? How?

- Plan: board starts with small living area, but can grow!
 - Start with 4x4 board
 - Pad all sides with zeros

0	1	1	1
1	0	0	1
0	0	0	1
0	0	1	1

0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	0	0	0	1	0
0	0	0	1	1	0
0	0	0	0	0	0

Unbounded? How?

- Plan: board starts with small living area, but can grow!
 - Start with 4x4 board
 - Pad all sides with zeros
 - Iterate forward one round

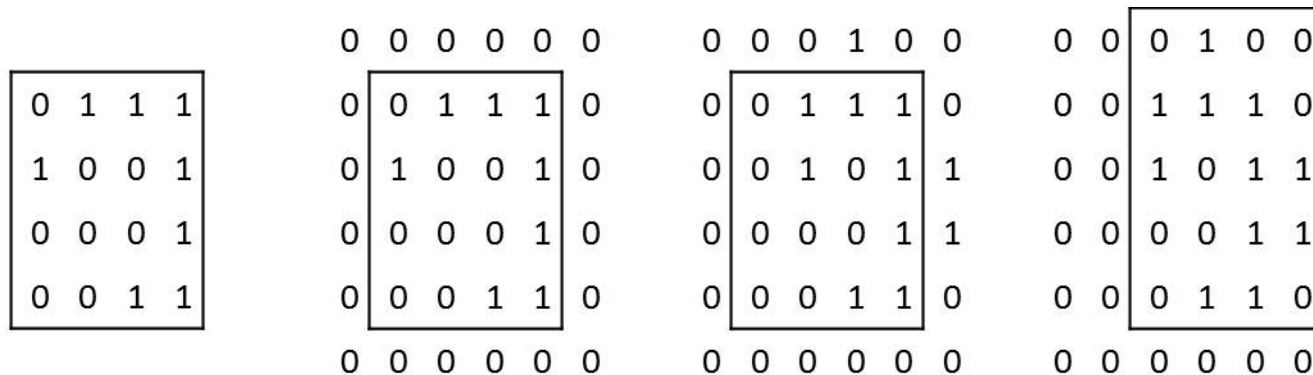
0	1	1	1
1	0	0	1
0	0	0	1
0	0	1	1

0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	0	0	0	1	0
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	1	0	0
0	0	1	1	1	0
0	0	1	0	1	1
0	0	0	0	1	1
0	0	0	1	1	0
0	0	0	0	0	0

Unbounded? How?

- Plan: board starts with small living area, but can grow!
 - Start with 4x4 board
 - Pad all sides with zeros
 - Iterate forward one round
 - Recalculate subboard with living cells



Unbounded? How?

- Plan: board starts with small living area, but can grow!
 - Start with 4x4 board
 - Pad all sides with zeros
 - Iterate forward one round
 - Recalculate subboard with living cells
 - (Un)Pad as necessary

0	1	1	1
1	0	0	1
0	0	0	1
0	0	1	1

0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	0	0	0	1	0
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	1	0	0
0	0	1	1	1	0
0	0	1	0	1	1
0	0	0	0	1	1
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	1	0	0
0	0	1	1	1	0
0	0	1	0	1	1
0	0	0	0	1	1
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	0	0	0
0	0	1	0	0	0
0	1	1	1	0	0
0	1	0	1	1	0
0	0	0	1	1	0
0	0	1	1	0	0
0	0	0	0	0	0

Unbounded? How?

- Plan: board starts with small living area, but can grow!
 - Start with 4x4 board
 - Pad all sides with zeros
 - Iterate forward one round
 - Recalculate subboard with living cells
 - (Un)Pad as necessary
 - Repeat

0	1	1	1
1	0	0	1
0	0	0	1
0	0	1	1

0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	0	0	0	1	0
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	1	0	0
0	0	1	1	1	0
0	0	1	0	1	1
0	0	0	0	1	1
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	1	0	0
0	0	1	1	1	0
0	0	1	0	1	1
0	0	0	0	1	1
0	0	0	1	1	0
0	0	0	0	0	0

0	0	0	0	0	0
0	0	1	0	0	0
0	1	1	1	0	0
0	1	0	1	1	0
0	0	0	1	1	0
0	0	1	1	0	0
0	0	0	0	0	0

Game of Life: Setting the Domain

```
//set the bounds  
var minLivingRow = 3;  
var maxLivingRow = 6;  
var minLivingColumn = 1;  
var maxLivingColumn = 4;
```

Game of Life: Setting the Domain

```
//set the bounds
var minLivingRow = 3;
var maxLivingRow = 6;
var minLivingColumn = 1;
var maxLivingColumn = 4;

//ranges for the board size
var boardRows = (minLivingRow-1)..(maxLivingRow+1);
var boardColumns = (minLivingColumn-1)..(maxLivingColumn+1);
```


Game of Life: Setting the Domain

```
//set the bounds
var minLivingRow = 3;
var maxLivingRow = 6;
var minLivingColumn = 1;
var maxLivingColumn = 4;

//ranges for the board size
var boardRows = (minLivingRow-1)..(maxLivingRow+1);
var boardColumns = (minLivingColumn-1)..(maxLivingColumn+1);

//domain of the game board
//this will change every iteration of the simulation!
var gameDomain: domain(2) = [boardRows, boardColumns];
```

Game of Life: Setting the Domain

```
//set the bounds
var minLivingRow = 3;
var maxLivingRow = 6;
var minLivingColumn = 1;
var maxLivingColumn = 4;

//ranges for the board size
var boardRows = (minLivingRow-1)..(maxLivingRow+1);
var boardColumns = (minLivingColumn-1)..(maxLivingColumn+1);

//domain of the game board
//this will change every iteration of the simulation!
var gameDomain: domain(2) = [boardRows, boardColumns];

//alive: 1; dead: 0
var lifeArray: [gameDomain] int;           //defaults to zeroes
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round  
//(0 means no life, 1 means life)  
proc lifeValueNextRound(x, y, currentBoard) {
```

```
}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round  
//(0 means no life, 1 means life)  
proc lifeValueNextRound(x, y, currentBoard) {
```

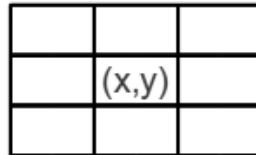
How can we just focus on the neighboring cells?

```
}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round  
//(0 means no life, 1 means life)  
proc lifeValueNextRound(x, y, currentBoard) {
```

How can we just focus on the neighboring cells?



```
}
```

Game of Life: Implementing Rules

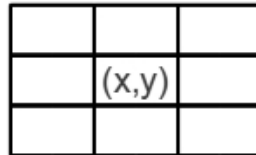
```
//returns whether there will be life at (x, y) next round
```

```
 //(0 means no life, 1 means life)
```

```
proc lifeValueNextRound(x, y, currentBoard) {
```

```
  //the 9 cells adjacent to (x, y)
```

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



```
}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round
```

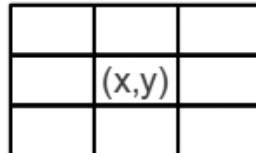
```
 //(0 means no life, 1 means life)
```

```
proc lifeValueNextRound(x, y, currentBoard) {
```

```
  //the 9 cells adjacent to (x, y)
```

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

How can we (easily) handle border cases?



```
}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round
```

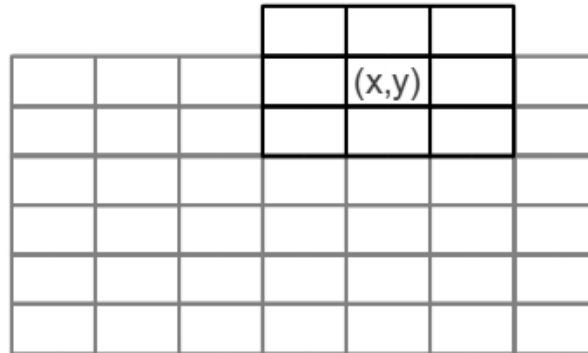
```
//(0 means no life, 1 means life)
```

```
proc lifeValueNextRound(x, y, currentBoard) {
```

```
  //the 9 cells adjacent to (x, y)
```

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

How can we (easily) handle border cases?

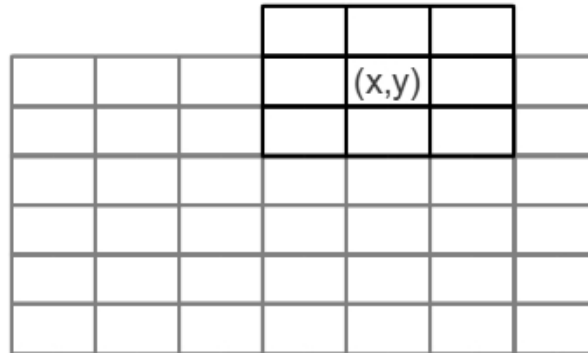


```
}
```


Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round
//(0 means no life, 1 means life)
proc lifeValueNextRound(x, y, currentBoard) {
  //the 9 cells adjacent to (x, y)
  var adjacentDomain : domain(2) = [x-1..x+1, y-1..y+1];

  //domain slicing!
  var neighborDomain = adjacentDomain [currentBoard.domain];
```



```
}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round
//(0 means no life, 1 means life)
proc lifeValueNextRound(x, y, currentBoard) {
    //the 9 cells adjacent to (x, y)
    var adjacentDomain : domain(2) = [x-1..x+1, y-1..y+1];

    //domain slicing!
    var neighborDomain = adjacentDomain [currentBoard.domain];

}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round
//(0 means no life, 1 means life)
proc lifeValueNextRound(x, y, currentBoard) {
  //the 9 cells adjacent to (x, y)
  var adjacentDomain : domain(2) = [x-1..x+1, y-1..y+1];

  //domain slicing!
  var neighborDomain = adjacentDomain [currentBoard.domain];
  var neighborSum = + reduce currentBoard[neighborDomain];
  neighborSum = neighborSum - currentBoard[x, y];

}
```

Game of Life: Implementing Rules

```
//returns whether there will be life at (x, y) next round
//(0 means no life, 1 means life)
proc lifeValueNextRound(x, y, currentBoard) {
    //the 9 cells adjacent to (x, y)
    var adjacentDomain : domain(2) = [x-1..x+1, y-1..y+1];

    //domain slicing!
    var neighborDomain = adjacentDomain [currentBoard.domain];
    var neighborSum = + reduce currentBoard[neighborDomain];
    neighborSum = neighborSum - currentBoard[x, y];

    //the survival/reproduction rules for the Game of Life
    if 2 <= neighborSum && neighborSum <= 3 && currentBoard[x, y] == 1 {
        return 1;
    } else if currentBoard[x, y] == 0 && neighborSum == 3 {
        return 1;
    } else { return 0; }
}
```

Game of Life: Supporting Boards

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Also, want to easily determine bounds on where life is! How?

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Also, want to easily determine bounds on where life is! How?

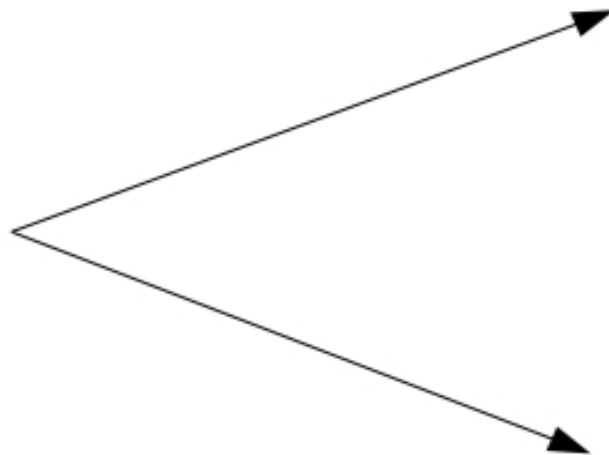
	6		9	
1	0	0	0	0
	0	0	0	0
	0	1	1	0
	1	1	0	0
5	0	1	0	0

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Also, want to easily determine bounds on where life is! How?

	6	9		
1	0	0	0	0
	0	0	0	0
	0	1	1	0
	1	1	0	0
5	0	1	0	0



	6	9		
1	0	0	0	0
	0	0	0	0
	0	3	3	0
	4	4	0	0
5	0	5	0	0

rows

	6	9		
1	0	0	0	0
	0	0	0	0
	0	7	8	0
	6	7	0	0
5	0	7	0	0

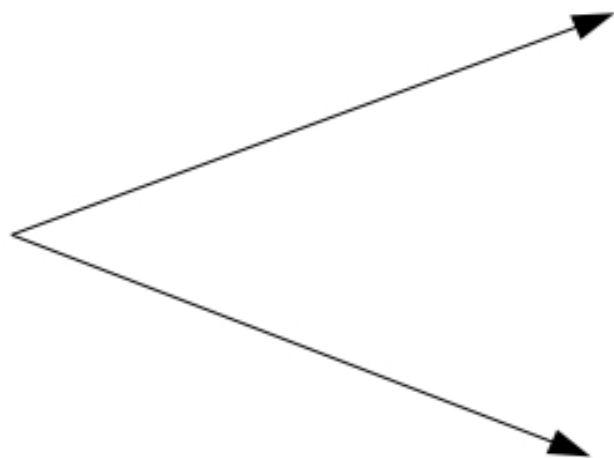
cols

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Also, want to easily determine bounds on where life is! How?

	6	9		
1	0	0	0	0
	0	0	0	0
	0	1	1	0
	1	1	0	0
5	0	1	0	0



	6	9		
1	0	0	0	0
	0	0	0	0
	0	3	3	0
	4	4	0	0
5	0	5	0	0

rows

rowIfAliveArray

	6	9		
1	0	0	0	0
	0	0	0	0
	0	7	8	0
	6	7	0	0
5	0	7	0	0

cols

colIfAliveArray

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Also, want to easily determine bounds on where life is! How?

```
maxLivingRow =  
    max reduce rowIfAliveArray;  
minLivingRow =  
    min reduce rowIfAliveArray;  
maxLivingColumn =  
    max reduce colIfAliveArray;  
minLivingColumn =  
    min reduce colIfAliveArray;
```

	6	9			
1	0	0	0	0	rows rowIfAliveArray
0	0	0	0		
0	3	3	0		
4	4	0	0		
5	0	5	0	0	

	6	9			
1	0	0	0	0	cols colIfAliveArray
0	0	0	0		
0	7	8	0		
6	7	0	0		
5	0	7	0	0	

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Doesn't work! Zeros!

```
maxLivingRow =  
    max reduce rowIfAliveArray;  
minLivingRow =  
    min reduce rowIfAliveArray;  
maxLivingColumn =  
    max reduce colIfAliveArray;  
minLivingColumn =  
    min reduce colIfAliveArray;
```

	6	9			
1	0	0	0	0	rows rowIfAliveArray
0	0	0	0		
0	3	3	0		
4	4	0	0		
5	0	5	0	0	

	6	9			
1	0	0	0	0	cols colIfAliveArray
0	0	0	0		
0	7	8	0		
6	7	0	0		
5	0	7	0	0	

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Doesn't work! Zeroes!

Solution: replace with middle index

```
maxLivingRow =  
    max reduce rowIfAliveArray;  
minLivingRow =  
    min reduce rowIfAliveArray;  
maxLivingColumn =  
    max reduce colIfAliveArray;  
minLivingColumn =  
    min reduce colIfAliveArray;
```

	6	9			
1	0	0	0	0	rows rowIfAliveArray
0	0	0	0		
0	3	3	0		
4	4	0	0		
5	0	5	0	0	

	6	9			
1	0	0	0	0	cols colIfAliveArray
0	0	0	0		
0	7	8	0		
6	7	0	0		
5	0	7	0	0	

Game of Life: Supporting Boards

```
//next turn's board  
var nextLifeArray: [gameDomain] int;
```

Doesn't work! Zeroes!

Solution: replace with middle index

```
maxLivingRow =  
    max reduce rowIfAliveArray;  
minLivingRow =  
    min reduce rowIfAliveArray;  
maxLivingColumn =  
    max reduce colIfAliveArray;  
minLivingColumn =  
    min reduce colIfAliveArray;
```

	6	9			
1	3	3	3	3	rows rowIfAliveArray
	3	3	3	3	
	3	3	3	3	
	4	4	3	3	
5	3	5	3	3	

	6	9			
1	7	7	7	7	cols colIfAliveArray
	7	7	7	7	
	7	7	8	7	
	6	7	7	7	
5	7	7	7	7	

Game of Life: Supporting Boards

```
//next turn's board
```

```
var nextLifeArray: [gameDomain] int;
```

```
//if life is here, it will contain its column index,
```

```
//otherwise, the board's middle column index
```

```
var columnIfAliveArray: [gameDomain] int;
```

```
//if life is here, it will contain its row index,
```

```
//otherwise, the board's middle row index
```

```
var rowIfAliveArray: [gameDomain] int;
```

Game of Life: Supporting Boards

```
//next turn's board
var nextLifeArray: [gameDomain] int;

//if life is here, it will contain its column index,
//otherwise, the board's middle column index
var columnIfAliveArray: [gameDomain] int;

//if life is here, it will contain its row index,
//otherwise, the board's middle row index
var rowIfAliveArray: [gameDomain] int;

...

//later on, use simple reductions:
maxLivingRow = max reduce rowIfAliveArray;
minLivingRow = min reduce rowIfAliveArray;
maxLivingColumn = max reduce columnIfAliveArray;
minLivingColumn = min reduce columnIfAliveArray;
```


Game of Life: Initial Life

	0					5
2	0	0	0	0	0	0
	0	0	1	1	1	0
	0	1	0	0	1	0
	0	0	0	0	1	0
	0	0	0	1	1	0
7	0	0	0	0	0	0

//default values are 0 (no life) and 1 (life)

//following locations start alive:

lifeArray[minLivingRow, minLivingColumn + 1] = 1;

lifeArray[minLivingRow, minLivingColumn + 2] = 1;

lifeArray[minLivingRow, minLivingColumn + 3] = 1;

lifeArray[minLivingRow + 1, minLivingColumn] = 1;

lifeArray[minLivingRow + 1, minLivingColumn + 3] = 1;

lifeArray[minLivingRow + 2, minLivingColumn + 3] = 1;

lifeArray[minLivingRow + 3, minLivingColumn + 2] = 1;

lifeArray[minLivingRow + 3, minLivingColumn + 3] = 1;

Game of Life: “If Alive” Functions

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of  
the middle row of array. */
```

```
proc rowIfAlive(x, y, array) {
```

```
}
```

Game of Life: “If Alive” Functions

- Easy: returning the row/column number

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of
   the middle row of array. */
proc rowIfAlive(x, y, array) {
    if array[x, y] == 1 {
        return x;
    }
}

}
```

Game of Life: “If Alive” Functions

- Easy: returning the row/column number
- Less easy: getting the index of the middle row

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of
   the middle row of array. */
proc rowIfAlive(x, y, array) {
    if array[x, y] == 1 {
        return x;
    }
}

}
```

Game of Life: “If Alive” Functions

- Easy: returning the row/column number
- Less easy: getting the index of the middle row
 - Use dim domain method to get 1-D subrange

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of
the middle row of array. */
proc rowIfAlive(x, y, array) {
  if array[x, y] == 1 {
    return x;
  }
  //determine and return the middle row index
  var rowRange = array.domain.dim(1);
}
}
```

Game of Life: “If Alive” Functions

- Easy: returning the row/column number
- Less easy: getting the index of the middle row
 - Use dim domain method to get 1-D subrange
 - Use high and low range properties

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of
the middle row of array. */
proc rowIfAlive(x, y, array) {
  if array[x, y] == 1 {
    return x;
  }
  //determine and return the middle row index
  var rowRange = array.domain.dim(1);
  var rowHigh = rowRange.high;
  var rowLow = rowRange.low;
}
```

Game of Life: “If Alive” Functions

- Easy: returning the row/column number
- Less easy: getting the index of the middle row
 - Use dim domain method to get 1-D subrange
 - Use high and low range properties
 - Calculate and return middle index

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of
the middle row of array. */
proc rowIfAlive(x, y, array) {
  if array[x, y] == 1 {
    return x;
  }
  //determine and return the middle row index
  var rowRange = array.domain.dim(1);
  var rowHigh = rowRange.high;
  var rowLow = rowRange.low;
  return (rowLow + rowHigh)/2;
}
```

Game of Life: “If Alive” Functions

- Easy: returning the row/column number
- Less easy: getting the index of the middle row
 - Use dim domain method to get 1-D subrange
 - Use high and low range properties
 - Calculate and return middle index
 - (Doesn't work if the range is strided.)

```
/* If life exists in array at location (x, y), then this returns the index of the row (x). Otherwise, this returns the index of
the middle row of array. */
proc rowIfAlive(x, y, array) {
  if array[x, y] == 1 {
    return x;
  }
  //determine and return the middle row index
  var rowRange = array.domain.dim(1);
  var rowHigh = rowRange.high;
  var rowLow = rowRange.low;
  return (rowLow + rowHigh)/2;
}
```


Game of Life: Main Loop

```
for round in 1..numRounds {
  forall (i , j) in gameDomain {           //set the elements of the next life array
    nextLifeArray[i,j] = lifeValueNextRound(i,j, lifeArray);
  }
  forall (i , j) in gameDomain {           //set the "location if alive" arrays
    rowIfAliveArray[i,j] = rowIfAlive(i,j, nextLifeArray);
    columnIfAliveArray[i,j] = columnIfAlive(i,j, nextLifeArray);
  }

  //reset the bounds with reductions
  maxLivingRow = max reduce rowIfAliveArray;
  minLivingRow = min reduce rowIfAliveArray;
  maxLivingColumn = max reduce columnIfAliveArray;
  minLivingColumn = min reduce columnIfAliveArray;

  //reset the game domain, including buffer of no life
  gameDomain = [(minLivingRow-1)..(maxLivingRow+1),
                (minLivingColumn-1)..(maxLivingColumn+1)];
  lifeArray = nextLifeArray;
}
```

Game of Life: Add writeln and Go!

- Add print statements for each iteration of the loop and watch it go
- I added a printLifeArray function
- Final version available at:

<https://dl.dropbox.com/u/43416022/SC12/GameOfLife.chpl>

Hands-on Session 2

[http://faculty.knox.edu/dbunde/
teaching/chapel/SC12/exercises.html](http://faculty.knox.edu/dbunde/teaching/chapel/SC12/exercises.html)

Using Chapel in the Classroom

Chapel in the Classroom

- Use in courses
 - Analysis of Algorithms
 - Programming Languages
 - Other courses?
- Hurdles
 - Still in development
- Discussion: How do you want to use Chapel?

Analysis of Algorithms

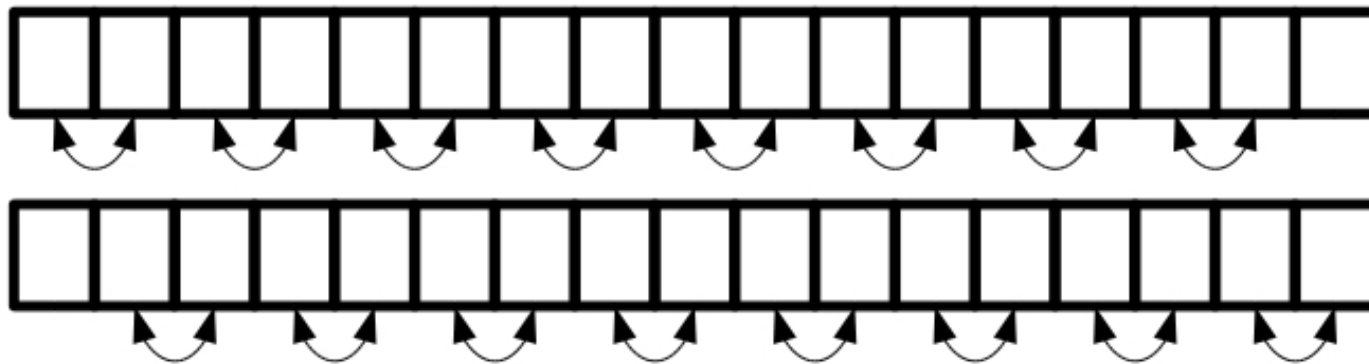
- Chapel material
 - Assign basic tutorial
 - Teach forall & cobegin (also algorithmic notation)
- Projects
 - Partition integers
 - BubbleSort
 - MergeSort
 - Nearest Neighbors

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: MergeSort

- Parallel divide-and-conquer: use cobegin
- Elegant division: split the Domain
- Speedup not as noticeable
- Example of expensive parallel overhead

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

Algorithms Takeaway

- Learning curve of Chapel is so low, students can start using parallelism very quickly

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
- Project:
 - Matrix Multiplication (two different ways)

PL: Thread Generation

- Ex. Java: have to create an object
- Chapel: instead create tasks
 - Chapel decides when to generate threads
 - Basic keyword: begin

```
begin {  
    producer.run();  
}
```

PL: Array Sum

- Divide between two tasks:

```
begin {
```

```
    // save value in lowerHalfSum
```

```
}
```

```
//loop to find upperHalfSum
```

```
total = lowerHalfSum + upperHalfSum
```

- Problem: new task might not finish in time
 - Solution: Chapel includes keyword 'sync'

PL: Synchronized Tasks

- Use sync:

```
sync {  
    begin {  
        //loop to find lowerHalfSum  
    }  
    begin {  
        //loop to find upperHalfSum  
    }  
}  
sum = lowerHalfSum + upperHalfSum
```

- Pattern used often; Chapel uses 'cobegin' to simplify.

PL: cobegin

- Use cobegin:

```
cobegin {  
    //loop to find lowerHalfSum  
    //loop to find upperHalfSum  
}
```

- Much simpler!

PL: forall

- “forall”: common command in parallel algorithm design
 - Give example
 - forall vs. coforall (data vs. task parallelism)
- Thread safety
 - Write arraySum with forall
 - Run it; get different results!
 - Define thread safe
 - Use 'sync' (for variables) to fix

PL: sync bottleneck and reduce

- sync causes a bottleneck:
 - Threads may block; Running time still linear!
- Reductions:
 - Divide-and-conquer solution
 - Simplify with 'reduce' keyword!

PL: Projects

- Matrix Multiplication
 - Did 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!
- Motivation is obvious
- Students love it!

How else might you use Chapel?

- Parallel Computing
 - Quick prototyping, easily-changed data distribution, ...
- 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

Disclaimer!

- Still in development
 - Error Messages thin
 - Recursive functions can't return arrays
 - Basic libraries missing
 - (Students thought this was awesome!)
- No Development Environment
 - Command-line compilation/running
 - Linux learning curve?

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?

Thanks!

dbunde@knox.edu

paithanq@gmail.com