Chapel: A Versatile Language for Teaching Parallel Programming

CCSCNE 2015: April 17
Kyle Burke
Plymouth State
kgburke@plymouth.edu
Acknowledgements

• Silent partner: David Bunde
• Material drawn from tutorials created with contributions from Johnathan Ebbers, Maxwell Galloway-Carson, Michael Graf, Ernest Heyder, Sung Joo Lee, Andrei Papanacea, 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
• Analysis of Algorithms
  — Parallel basics
• Hands-On Time
• Programming Languages
  — More details
  — Game of Life
• Hands-On Time
• Wrap Up
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: \( x = + \text{ reduce } A \)  //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

• Function promotion:
  \[ B = f(A); \]  //applies \( f \) elementwise for any function \( f \)

• Includes built-in operators:
  \[ C = A + 1; \]
  \[ D = A + B; \]
  \[ E = A \times B; \]
  ...

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://turing.plymouth.edu/~kgb1013/chapelWorkshopHandsOn.php

• 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) \textcolor{red}{then} y = 3; \textcolor{red}{else} y = 1;
  while(x < 5) \textcolor{red}{do} x++;
Example: Reading until eof

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

proc addOne(x: int) : int {
    y = x + 1;
    return y;
}

parameter type

return type

typte

Arrays

• Indices determined by a range:
  var A : [0..4] 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:

• 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 {
    writeln(i);
  }

• Can also use array or anything iterable
Using Chapel in your courses
Analysis of Algorithms

• **No** class time
• Teach parallel with sequential algorithms.
• Parallelism needed:
  – parallel loop (forall in alg. Notation)
  – parallel-do (divide and conquer)
• Use only two parallel keywords
  – forall
  – cobegin
Analysis of Algorithms

• Projects: (sequential & parallel)
  – List partition
  – BubbleSort
  – MergeSort
  – NearestNeighbors
forall loops

var total : int = 0;
forall i in 1..100000 with (ref total) {
    total = total + i;
}
writeln("total: ", total); //5000050000
forall loops

var total : int = 0;
forall i in 1..100000 with (ref total) {
    total = total + i;
}
writeln("total: ", total); //5000050000

Why doesn’t this work?
forall loops

var total : sync int = 0;
forall i in 1..100000 with (ref total) {
    total = total + i;
}

var nonSyncTotal = total;
writeln("total: ", nonSyncTotal); //5000050000
Algorithms Project: BubbleSort

Instead of left-to-right, test all pairs in two steps!

Two nested forall loops
(in sequence) inside a for loop
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)
    }
}
cobegin

var total : int = 0;
for i in 1..100 {
    total += i;
}
writeln(“total: “, total);
cobegin

var total : int = 0;
for i in 1..100 {
    total += i;
}
writeln("total: ", total);

Let's do it half-by-half!
cobegin

var lowTotal : int = 0;
var highTotal : int = 0;
for i in 1..50 {
    lowTotal += i;
}
for i in 51..100 {
    highTotal += i;
}
var total = lowTotal + highTotal;
writeln("total: ", total);
cobegin

var lowTotal : int = 0;
var highTotal : int = 0;
for i in 1..50 {
    lowTotal += i;
}
for i in 51..100 {
    highTotal += i;
}
var total = lowTotal + highTotal;
writeln("total: ", total);
cobegin

var lowTotal : int = 0;
var highTotal : int = 0;
for i in 1..50 do lowTotal += i;
for i in 51..100 do highTotal += i;
var total = lowTotal + highTotal;
writeln("total: ", total);
cobegin

var lowTotal : int = 0;
var highTotal : int = 0;
for i in 1..50 do lowTotal += i;
for i in 51..100 do highTotal += i;
var total = lowTotal + highTotal;
writeln("total: ", total);

Let's do them in parallel!
cobegin

var lowTotal : int = 0;
var highTotal : int = 0;
cobegin {
    for i in 1..50 do lowTotal += i;
    for i in 51..100 do highTotal += i;
}
var total = lowTotal + highTotal;
writeln(“total: “, total);
cobegin

var lowTotal : int = 0;
var highTotal : int = 0;
cobegin with (ref lowTotal, ref highTotal) {
    for i in 1..50 do lowTotal += i;
    for i in 51..100 do highTotal += i;
}
var total = lowTotal + highTotal;
writeln("total: ", total);
Algorithms Project: MergeSort

• Parallel divide-and-conquer: use cobegin
Algorithms Project: MergeSort

• Parallel divide-and-conquer: use cobegin

\[
\begin{array}{cccccccccccc}
12 & 8 & 5 & 15 & 7 & 4 & 4 & 0 & 16 & 7 & 1 & 9 \\
\end{array}
\]
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

\[
\begin{array}{cccccccc}
12 & 8 & 5 & 15 & 7 & 4 & 4 & 0 & 16 & 7 & 1 & 9 \\
\end{array}
\]
### Algorithms Project: MergeSort

- Parallel divide-and-conquer: use cobegin

<table>
<thead>
<tr>
<th>12</th>
<th>8</th>
<th>5</th>
<th>15</th>
<th>7</th>
<th>4</th>
<th>4</th>
<th>0</th>
<th>16</th>
<th>7</th>
<th>1</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>
More Algorithms Projects

Other projects:
• List-partition
  • Brute-force
  • Can enumerate all possibilities, then forall.
More Algorithms Projects

Other projects:
• List-partition
  • Brute-force
  • Can enumerate all possibilities, then forall.

• Nearest-Neighbors
  • 2-D point set
  • Brute force w/forall
  • Divide-and-conquer: tough!
Algorithms Takeaway

• Chapel is so easy, students pick it up on their own.

• Facilitates parallel algorithms in traditional course.
Hands-on time

(and/or break)

http://turing.plymouth.edu/~kgb1013/chapelWorkshopHandsOn.php
Programming Languages

• “HPC Paradigm”
• Speedup on first day!
• Teach lots of features
  – Parallel syntax
  – Scientific computing (domains, promotion)
• Language is still in production. Students can join the mailing lists, etc.
Programming Languages

• Discuss design choices!
• Projects:
  − Boolean List XOR
  − Matrix Multiplication
  − Game of Life
    • Unbounded board size
    • No explicit array copying
  − Collatz Conjecture Testing
var total = 0;
for i in 1..100 do total += i;

writeln('Sum is ', total, '.');
PL: Task Generation

```plaintext
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

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

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

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.');
```

We can add a Timer to measure running time!
We can add a Timer to measure running time!

```plaintext
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.');
```
We can add a Timer to measure running time!

```plaintext
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.');
```
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.');
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;

timer.start();

for i in 1..100 do total += i;

timer.stop();

writeln('Sum is ', total, '.');
writeln('That took ', timer.elapsed(), ' seconds.');
use Time;
var timer: Timer;
var total = 0;
var highTotal = 0;
var lowTotal = 0;
timer.start();
begin with (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.');
**PL: Task Generation**

**Does it work?**

```plaintext
use Time;
var timer: Timer;
var total = 0;
var highTotal = 0;
var lowTotal = 0;
timer.start();
begin with (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.');
```
PL: Synchronization

No: top thread may not finish.
Solution: sync
PL: Synchronization

No: top thread may not finish.
Solution: sync

```plaintext
sync {
    begin {
        ...
    }
    begin {
        ...
    }
}
```
Use sync:

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

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

total = lowTotal + highTotal;
...
```
PL: Syntactic Sugar

Ask students: How common is this?

```plaintext
sync {
    begin {
        //single line of code
    }
    begin {
        //another single line
    }
    ...
    begin {
        //even yet another single line
    }
}
```
PL: Syntactic Sugar

Ask students: How common is this?

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

What did Chapel designers do?
PL: Syntactic Sugar

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

forall: data-parallel loop

```plaintext
var sum = 0;
forall i in 1..100 with (ref sum) {
    sum += i;
}
writeln("Sum is: ", sum, ".");
```
forall: data-parallel loop

```pl
var sum = 0;
forall i in 1..100 with (ref sum) {
    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

```plaintext
var sum : sync int;
sum = 0;
forall i in 1..100 with (ref sum) {
    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

8  -1  2  21  1  -5  12  3
PL: Reductions

```
  30
 /   \
7   23
|     |
8   -1   2   21   1   -5   12   3
|     |     |     |     |     |
-1   2   -5   12   3
```
PL: Reductions

41

30

7 23

8 -1 2 21 1 -5 12 3

11

-4

15
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

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

```plaintext
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?

```javascript
var sums = + scan array;
```

```
8  -1  2  21  1  -5  12  3
```
PL: sync bottleneck and reduce

All intermediate values?

```plaintext
var sums = + 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!

```plaintext
var factorials = * scan (1..10);

var threes : [1..10] int;
forall i in threes.domain do
    threes[i] = 3;
var powersOfThree = * scan threes;
```
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 ↔ Motivates Syntax

• Students love it!
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 to Living if exactly 3 living neighbors
  – Living cell to 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
Unbounded? How?

- Plan: board starts with small living area, but can grow!
  - Start with 4x4 board
  - Pad all sides with zeros
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
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 sub-board 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 sub-board 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 1 1
0 0 0 0
0 0 0 0
```
```
//set the bounds
var minLivingRow = 3;
var maxLivingRow = 6;
var minLivingColumn = 1;
var maxLivingColumn = 4;
//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);
G.o.L.: 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];
G.o.L.: 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) {
    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];

    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: 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?

   \[
   \begin{array}{ccc}
   & & \\
   & (x,y) & \\
   & & \\
   \end{array}
   \]

}
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];
   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: 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[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;
  }
}

How can we (easily) handle border cases?

```plaintext
(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];
    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;
    }
}

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];

    // (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];

  if 2 <= neighborSum && neighborSum <= 3 && currentBoard[x, y] == 1 {
    return 1;
  } else if currentBoard[x, y] == 0 && neighborSum == 3 {
    return 1;
  } else {
    return 0;
  }
}
//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: 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

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

How do we find the bounds for the next board?

```
<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
// next turn's board
var nextLifeArray: [gameDomain] int;

1 0 0 0 0
0 0 0 0
0 1 1 0
1 1 0 0
5 0 1 0 0

6 9
5

0 0 0 0
0 0 0 0
0 3 3 0
4 4 0 0
0 5 0 0

6 9
Game of Life: Supporting Boards

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

```
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
3 3 3 3
3 3 3 3
3 3 3 3
4 4 3 3
3 5 3 3
5
```
Game of Life: Supporting Boards

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

```plaintext
1   0   0   0   0
0   0   0   0   0
0   1   1   0   0
1   1   0   0   0
5   0   1   0   0
6   9
```

```plaintext
1   3   3   3   3   3   3   3   3   3
3   3   3   3   3   3   3   3   3   3
3   3   3   3   3   3   3   3   3   3
4   4   3   3   3   3   3   3   3   3
5   3   5   3   3   3   3   3   3   3
rowIfAliveArray
```
Game of Life: Supporting Boards

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

```
1 0 0 0 0
0 0 0 0
0 1 1 0
1 1 0 0
```

```
6 9
```

```
1 3 3 3 3
3 3 3 3
3 3 3 3
4 4 3 3
5 3 5 3 3
```
rowIfAliveArray

```
6 9
```

```
1 7 7 7 7
7 7 7 7
7 7 8 7
6 7 7 7
5 7 7 7 7
```
colIfAliveArray
Game of Life: Supporting Boards

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

maxLivingRow =
    max reduce rowIfAliveArray;
minLivingRow =
    min reduce rowIfAliveArray;
maxLivingColumn =
    max reduce colIfAliveArray;
minLivingColumn =
    min reduce colIfAliveArray;
```
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: “If Alive” Functions

• Easy: returning the row/column number

```plaintext
proc rowIfAlive(row, column, array) {
    if array[row, column] == 1 {
        return row;
    }
}
```
Game of Life: “If Alive” Functions

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

```
proc rowIfAlive(row, column, array) {
    if array[row, column] == 1 {
        return row;
    }
    // determine and return the middle row index
}
```
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

```plaintext
proc rowIfAlive(row, column, array) {
    if array[row, column] == 1 {
        return row;
    }
    //determine and return the middle row index
    var rowRange = array.domain.dim(1); //dim(2) for columns
}
```
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

```javascript
proc rowIfAlive(row, column, array) {
    if array[row, column] == 1 {
        return row;
    }
    //determine and return the middle row index
    var rowRange = array.domain.dim(1); //dim(2) for columns
    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

```javascript
proc rowIfAlive(row, column, array) {
  if array[row, column] == 1 {
    return row;
  }
  //determine and return the middle row index
  var rowRange = array.domain.dim(1); //dim(2) for columns
  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 time

(and/or break)

http://turing.plymouth.edu/~kgb1013/chapelWorkshopHandsOn.php
Features for distributed memory
Representing locality

- Give control over where code is executed:
  ```
  on Locales[0] 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

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

```plaintext
var A : [D] int dmapped Cyclic(startIdx=1)
```
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!
kgburke@plymouth.edu