High-level parallel programming using Chapel

David Bunde, Knox College
Kyle Burke, Colby College

Nov 21, 2013
Acknowledgements

• 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
• 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
  – Why Chapel?
  – Algorithms
  – Hands on time

• Part II: 3:30-5:00
  – Programming languages
  – Parallel programming
  – Hands on time
  – Summary / discussion
Basic Facts about Chapel

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

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

• Supports scripting-like programs:
  writeln(“Hello World!”);

• Also provides objects and modules
Provides High-level Operations

• Reductions
  Ex: x = + reduce A  //sets x to sum of elements of A
  Also valid for other operators (min, max, *, ...)

• Scans
  Like a reduction, but computes value for each prefix
  A = [1, 3, 2, 5];
  B = + scan A;  //sets B to [1, 1+3=4, 4+2=6, 6+5=11]
Provides High-level Operations (2)

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

• 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
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/SC13/

• 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)
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-1.8 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
Algorithms:
Easy implementation of parallelism
Using Chapel in Algorithms

• Give students a quick (~1 lecture) introduction to Chapel syntax and provide tutorials
• Teach what you need - goal is not language coverage
“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

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

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

• 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 Loops

• Two kinds of parallel loops:
  for all 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

• 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:

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

  ```plaintext
  cobegin {
    statement1;
    statement2;
    ...
  }
  ```
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 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)
}
Algorithms Project: BubbleSort

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

\[ \lim_{p \to n} T(n,p) = O(n) \]
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

\[ 12, 8, 5, 15, 7, 4, 4, 0, 16, 7, 1, 9 \]

\[ 4, 5, 7, 8, 12, 15 \quad 0, 1, 4, 7, 9, 16 \]
Algorithms Project: MergeSort

Parallel divide-and-conquer: use cobegin

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

\[
\begin{array}{ccccccc}
4 & 5 & 7 & 8 & 12 & 15 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 4 & 7 & 9 & 16 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 4 & 4 & 5 & 7 & 7 & 8 & 9 & 12 & 15 & 16 \\
\end{array}
\]
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: Reductions
Summing values in an array

| 2 | 1 | 4 | 3 | 1 | 3 | 0 | 2 |
Summing values in an array

```
[2, 1, 4, 3, 1, 3, 0, 2]
```
Summing values in an array

2 1 4 3 1 3 0 2
Finding max of an array

![Binary Tree Diagram]
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 1 value to tally
Parallel reduction framework

Tally: Intermediate state of computation
i = Init: Create "empty" tally
a = Accumulate: Add 1 value to tally
c = Combine: Combine 2 tallies
rg = Reduce−gen: Generate result from tally
Defining reductions

• Tally: Intermediate state of computation

• Combine: Combine 2 tallies

• Reduce-gen: Generate result from tally

• Init: Create “empty” tally

• Accumulate: Add 1 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 1 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 1 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 1 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 1 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?
Reducions in Chapel

• Express reduction operation in single line:
  var s = + reduce A;  //A is array, s gets sum

• Supports +, *, ^ (xor), &&, ||, max, min, ...

• minloc and maxloc return a tuple with value and its index:
  var (val, loc) = minloc reduce A;
Reduction example

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

```typescript
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; }
}
And that’s not all... (scans)

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

<table>
<thead>
<tr>
<th>A</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
And that’s not all… (scans)

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

<table>
<thead>
<tr>
<th>A</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

- Useful answering queries like
  “What is the sum of elements 2 thru 7?”
  \[ \text{sum[7]} - \text{sum[1]} \]
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
Downward pass with function labels

\[ i = \text{init} \]
\[ a = \text{accumulate} \]
Presenting reductions

• Using reductions with standard functions
  – Optionally including scans

• Defining your own reductions
First hands on time

http://faculty.knox.edu/dbunde/teaching/chapel/SC13/exercises.html
Programming languages
Programming Languages

• High-Performance Computing as Paradigm

• Lots of design choices in Chapel to discuss:
  – Task Creation (instead of Threads) with 'begin'.
  – Task Synchronicity with 'sync' and cobegin
  – Parallel loops: forall and coforall
  – Thread safety using variable 'sync'
  – reduce overcomes bottleneck
PL: Task Generation

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

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

We can add a Timer to measure running time!
PL: Task Generation

```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!

```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.');
```
PL: Task Generation

Now let's use another thread!

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

writeln('Sum is ', total, '.');
writeln('That took ', timer.elapsed(), ' seconds.');</plaintext>

Note: ref(highTotal) at begin
PL: Task Generation

Now let's use another thread!

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

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

Result: faster, but sometimes incorrect.
```
PL: Synchronization

Incorrect: top thread may not finish.

Chapel provides a solution: sync

```plaintext
sync {
    begin {
        ...
    }
    begin {
        ...
    }
    ...
}
```
PL: Synchronization

Use sync:

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

Ask students: How common is this?

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

So, what did language designers do?
PL: Syntactic Sugar

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

PL: forall

forall: data-parallel loop

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

forall: data-parallel loop

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

Ask: Why doesn't this work?
PL: HPC Concepts

• Why doesn't it work?
  – Race conditions
  – Atomicity
  – Synchronization solutions
PL: forall

One solution: synchronized variables

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

• sync causes a bottleneck:
  – Running time still technically linear.

• Reductions:
  – Divide-and-conquer solution
  – Simplify with 'reduce' keyword!
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!
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

```javascript
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

Array (hierarchy)

Array Domain (indices) — Array Values
Chapel Domains

Array (hierarchy)

(Sparse)

Domain

Array Values
Chapel Domains

Array (hierarchy)

0, 2, 4, 6, ..., 6000

(Range)

Domain

Array Values
Chapel Domains

Array (hierarchy)

2-D
0, 2, 4, 6, ..., 6000
1,
2,
3,
..., 50
(2 Ranges)

Domain

Array Values
Chapel Domains

Array (hierarchy)

0, 2, 4, 6, ..., 6000

(Combo)

Domain

Array Values
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)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>
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:

    ```
    var R : range = 1..10;
    var A : [R] int;
    R = 0..10;       //no effect on array
    A[0] = 5;        //runtime error
    ```

    ```
    var D : domain(1) = {1..10};
    var A : [D] int;
    D = 0..10;       //resizes array
    A[0] = 5;        //ok
    ```

- Domains are more general; some are not sets of integers
Domain Slices (Intersection)

domain0: {0..2, 1..3}
domain1: {1..3, 3..5}
Domain Slices (Intersection)

domain0: {0..2, 1..3}
domain1: {1..3, 3..5}

domain2: {1..2, 3..3}
Domain Slices (Intersection)

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

domain0: {0..2, 1..3}
domain1: {1..3, 3..5}

domain2: {1..2, 3..3}
Domain Slices (Intersection)

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

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

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 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
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
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?

    (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};

    (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};

    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];

    (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];
}
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
5 0 1 0 0
4 0 0 0 0
3 0 0 0 0
2 0 1 1 0
1 0 1 1 0
0 1 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?
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?

maxLivingRow =
    max reduce rowIfAliveArray;
minLivingRow =
    min reduce rowIfAliveArray;
maxLivingColumn =
    max reduce colIfAliveArray;
minLivingColumn =
    min reduce colIfAliveArray;
Game of Life: Supporting Boards

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

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

Doesn't work! Zeros!
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;

// rows
// colIfAliveArray
// cols
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;
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

//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

```plaintext
/* 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;
    }
    return x;
}
```
Game of Life: “If Alive” Functions

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

```javascript
/* 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

```c
/* 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/SC13/GameOfLife.chpl
Parallel programming
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
Much harder than I thought

• Even a strong student struggled with code that sent messages to another instance of itself
  – Seemed like challenge of distributed memory environment
  – Weak OO background?
Global-view

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

```javascript
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:
  ```java
  on Locales[0] do
    something();
  ```
- and where data is placed:
  ```java
  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

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

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

- Lots of stuff on Chapel website
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
Second hands on time

http://faculty.knox.edu/dbunde/teaching/chapel/SC13/exercises.html
Summary / discussion
How else might you use Chapel?

• Operating Systems
  – Easy thread generation for scheduling projects

• Software Design
  – Some parallel design patterns have lightweight Chapel implementations

• Artificial Intelligence
  (or other courses w/ computationally-intense projects)

• Independent Projects
Caveats

• Still in development
  – Error messages thin
  – New versions every 6 months
  – Not many libraries
  – (Students thought this was awesome!)

• No development environment
  – Command-line compilation in Linux
Conclusions

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

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

dbunde@knox.edu
paithanq@gmail.com