An Overview of Chapel

Jonathan Turner
University of Colorado at Boulder
(Cray Summer Intern 2010)
Before we get started

- This talk is an overview of some of Chapel’s unique features, later talks will go into more detail

- Disclaimer: This presentation assumes the newest, yet unreleased, Chapel syntax, so there are keyword differences with the latest released version of the language
What is Chapel?

- Modern distributed, parallel (and concurrent) language
  - Uses partitioned global address space (PGAS) as its communication layer
  - Includes affinity control

- Compiled, general purpose language
  - Has many of the conveniences of a dynamic language

- Has first-class support for multidimensional arrays, as well as sparse and strided arrays

- Open-source implementation
  - Available now at https://sourceforge.net/projects/chapel/
What are Chapel’s Goals?

- **Main Goal: Improve programmer productivity**
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Provide better portability than current programming models
  - Improve robustness of parallel codes

- Be familiar to the seasoned Fortran/C/C++ developer as well as the new crop of Java/Matlab/Python programmers
  - Allows both OOP and traditional imperative styles

- Allow users to work with algorithms and concurrency at various levels of detail
A source-driven

OVERVIEW OF CHAPEL
Generics

\[
\text{proc add_or_concat}(x, y) \{ \\
    \text{return } x + y; \\
\}
\]

\[
\text{writeln(add_or_concat(3, 4));} \\
\text{writeln(add_or_concat("3", "4"));}
\]

- One function can mean many things
- Compiler works with what you call the function with
- The example prints the number 7 followed by the string “34”
Tuples

var y: 3*int = (4, 5, 6);

var (a, b, c) = y;
//assigns a=4, b=5, c=6

var d = ((...y), (...y));
//d = (4, 5, 6, 4, 5, 6)

var z = y + y;
//z = (8, 10, 12)

- Tuples are a powerful first-class citizen in Chapel
- Allow for destructuring, expansion, and a number of operations (including lexicographical ordering)
Iterators

```python
iter squares(n:int) : int { 
    for i in 1..n do 
        yield i * i;
 }

for s in squares(10) { 
    writeln(s);
 }
```

- Iterators are like functions but produce a “stream” of values
- Can be used in `for` statements
...AND THEN THE FUN BEGINS
Task Parallelism: Sync and Begin

sync {  
begin treeSearch(root);  
}

proc treeSearch(node) {  
if node == nil then return;  
begin treeSearch(node.right);  
begin treeSearch(node.left);  
}
Forall and Coforall

\texttt{for x in 1..n do}
\begin{verbatim}
  expensive_operation(x);
\end{verbatim}

\texttt{forall x in 1..n do}
\begin{verbatim}
  expensive_operation(x);
\end{verbatim}

\texttt{coforall x in 1..n do}
\begin{verbatim}
  expensive_operation(x);
\end{verbatim}

- Like \texttt{for}, but now each iteration can be done in parallel
- \texttt{forall} hints that each iteration can be done in parallel using a “recipe” but the loop must be serializable
- \texttt{coforall} requires that each iteration be done in parallel
A New Way of Looking at Indices

Traditional

0 Value
1 Value
2 Value

Chapel

Domain

Array Values
Domains

```plaintext
var D: domain(1) = [1..1000];
var a: [D] int;
var b: [D] string;

a[5] = 10;
b[5] = "test";

var D: domain(2) = [1..m, 1..n];

var Inner: subdomain(D) = [2..m-1, 2..n-1];

var Strided = D by (2, 2);
```

- Abstract index sets
- For example: 1-based or 0-based, you pick
- Can be strided, multi-dimensional
- Can also span multiple machines (which we’ll see later)
- First example can also be written: var a: [1..1000] int
Domains Come in Many Shapes

```
var D: domain(2) = [1..m, 1..n];

var Strided = D by (2, 4);

var Sparse: sparse subdomain(D);

var Vertices: domain(Vertex);

var Names: domain(string);
```
…and Support Set-like Operations

Domain Intersection
Associative Domains

```typescript
var D: domain(int);
D.add(3);
D.add(5);

var a: [D] int;
var b: [D] string;

a[3] = 100;
b[5] = “test”;

var D2: domain(Shape);
D2.add(triangle);
D2.add(square);
```

- We can now handle arbitrary indices, not just ones based on a range of values
- Work like other arrays, we can iterate through it just as simply
- Also works with any value type, not just integers and integers
Domain Maps

use CyclicDist;

const tpl = 2;

var myCyclicDist =
    new dmap(new Cyclic(
        startIdx=(1,1)
        dataParTasksPerLocale=tpl));

var dom:
    domain(2) dmapped myCyclicDist =
    [1..n, 1..n];

• Domain maps allow us to connect our domain to multiple processing units
• Domain maps can pick from a variety of distribution methods (Cyclic distribution shown)
• Users can create their own distributions
Summary

- Chapel has a lightweight, familiar syntax
- It has powerful abstractions that let us handle arrays and their indices in new ways
- These abstractions allow the user to focus on the problem and the distribution separately
Thank You.

Questions?