Chapel: Data Parallelism
Data Parallelism:
- parallelism is driven by collections of data
  - data aggregates (arrays)
  - sets of indices (ranges, domains)
  - other user-defined collections
- e.g., “for all elements in array A ...”

Task Parallelism:
- parallelism is expressed using distinct tasks
- e.g., “create a task to do foo() while another does bar()”

(Of course, data parallelism is executed using tasks and task parallelism typically operates on data, so the line can get fuzzy at times...)
Data Parallel Hello World

```
config const numIters = 100000;

forall i in 1..numIters do
  writeln("Hello, world! ",
          "from iteration ", i,
          " of ", numIters);
```
Outline

- Domains and Arrays
  - Rectangular Domains and Arrays
  - Iterations and Operations
- Other Domain Types
- Reductions
Domains

**Domain:** A first-class index set

- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s *region* concept
- Domains may optionally be distributed
config const \( m = 4, \ n = 8; \)

\texttt{var D: domain(2) = [1..m, 1..n];}
config const m = 4, n = 8;

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

var InnerD: subdomain(D) = [2..m-1, 2..n-1];
Domains Define Arrays

- **Syntax**
  
  \[
  \text{array-type: } [ \text{domain-expr} ] \text{ elt-type}
  \]

- **Semantics**
  - Stores element for each index in \textit{domain-expr}

- **Example**
  
  ```chapel
  var A, B: [D] real;
  
  A
  
  B
  ```

- **Revisited example**
  
  ```chapel
  var A: [1..3] int; // creates anonymous domain [1..3]
  ```
Domain Iteration

• For loops (discussed already)
  • Execute loop body once per domain index, serially

```chapel
for i in InnerD do ...
```

• Forall loops
  • Executes loop body once per domain index, in parallel
  • Loop must be *serializable* (executable by one task)

```chapel
forall i in InnerD do ...
```

• Loop variables take on `const` domain index values
### Other Forall Loops

Forall loops also support...

- **A shorthand notation:**
  
  ```chapel
  [(i,j) in D] A(i,j) = i + j/10.0;
  ```

- **Expression-based forms:**
  
  ```chapel
  A = forall (i,j) in D do i + j/10.0;
  ```
  
  ```chapel
  A = [(i,j) in D] i + j/10.0;
  ```
Domain values support...

- **Methods for creating new domains**

  ```
  var D2 = InnerD.expand(1,0);
  ```

  ```
  var D3 = InnerD.translate(0,1);
  ```

- **Intersection via Slicing**

  ```
  var D4 = D2[D3];
  ```
Array Slicing/Sub-Arrays

Indexing into arrays with domain values results in a sub-array expression

\[ A[\text{InnerD}] = B[\text{InnerD}.\text{translate}(0,1)]; \]

A \quad = \quad B
Array Reallocation

Reassigning a domain logically reallocates its arrays

- values are preserved for common indices

\[
D = [1..2\cdot m, 1..2\cdot n];
\]

![Diagram of arrays A and B](chart.png)
Array Iteration

- Array expressions also support for and forall loops

```chapel
for a in A[InnerD] do ...
```

- Array loop variables refer to array values (modifiable)

```chapel
forall a in A[InnerD] do ...
```

```chapel
forall (a, (i,j)) in (A, D) do a = i + j/10.0;
```
Arrays can be indexed using variables of their domain’s index type (e.g., tuples) or lists of integers.

```chapel
var i = 1, j = 2;
var ij = (i,j);
A[ij] = 1.0;
A[i, j] = 2.0;

Array indexing can use either parentheses or brackets.

A(ij) = 3.0;
A(i, j) = 4.0;
```
Functions/operators expecting scalars can also take arrays, causing each element to be passed in...
forall loops are implemented using multiple tasks
  • details depend on what is being iterated over
so are operations that are equivalent to forall loops
  • promoted operators/functions, whole array assignment, ...

many times, this parallelism can seem invisible
  • for this reason, Chapel’s data parallelism can be considered
    *implicitly parallel*
  • it also tends to make the data parallel features easier to use
    and less likely to result in bugs as compared to explicit tasks
Outline

- Domains and Arrays
- Other Domain Types
- Reductions
- NAS MG Stencil Revisited
Chapel supports several domain types...

```chapel
var OceanSpace = [0..#lat, 0..#long],
    AirSpace = OceanSpace by (2,4),
    IceSpace: sparse subdomain(OceanSpace) = genCaps();
```

```
var Vertices: domain(opaque) = ..., People: domain(string) = ...;
```
All domain types can be used to declare arrays...

```chapel
var Ocean: [OceanSpace] real,
    Air: [AirSpace] real,
    IceCaps[IceSpace] real;
```

```chapel
var Weight: [Vertices] real,
    Age: [People] int;
```

Chapel: Data Parallelism
...to iterate over index sets...

forall \ ij \ in \ AirSpace \ do
\ Ocean(\ ij)\ +=\ IceCaps(\ ij); 

forall \ v \ in \ Vertices \ do
\ Weight(\ v)\ =\ numEdges(\ v); 
forall \ p \ in \ People \ do
\ Age(\ p)\ +=\ 1;
Slicing

...to slice arrays...

\[ \text{Ocean}[\text{AirSpace}] += \text{IceCaps}[\text{AirSpace}] \];

...Vertices[Interior]...  ...People[Interns]...

Chapel: Data Parallelism
Reallocation

...and to reallocate arrays

```chapel
AirSpace = OceanSpace by (2,2);
IceSpace += genEquator();
```

```
newnode = Vertices.create(); People += "vass";
```
Associative Domains and Arrays by Example

```chapel
var Presidents: domain(string) =
    ("George", "John", "Thomas", "James", "Andrew", "Martin");

Presidents += "William";

var Age: [Presidents] int,
    Birthday: [Presidents] string;

Birthday("George") = "Feb 22";

forall president in President do
    if Birthday(president) == today then
        Age(president) += 1;
```

<table>
<thead>
<tr>
<th>Presidents</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>277</td>
</tr>
<tr>
<td>John</td>
<td>274</td>
</tr>
<tr>
<td>Thomas</td>
<td>266</td>
</tr>
<tr>
<td>James</td>
<td>251</td>
</tr>
<tr>
<td>Andrew</td>
<td>242</td>
</tr>
<tr>
<td>Martin</td>
<td>227</td>
</tr>
<tr>
<td>William</td>
<td>236</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Birthday</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feb 22</td>
<td>277</td>
</tr>
<tr>
<td>Oct 30</td>
<td>274</td>
</tr>
<tr>
<td>Apr 13</td>
<td>266</td>
</tr>
<tr>
<td>Mar 16</td>
<td>251</td>
</tr>
<tr>
<td>Mar 15</td>
<td>242</td>
</tr>
<tr>
<td>Dec 5</td>
<td>227</td>
</tr>
<tr>
<td>Feb 9</td>
<td>236</td>
</tr>
</tbody>
</table>
Outline

- Domains and Arrays
- Other Domain Types
- Reductions
- NAS MG Stencil Revisited
Reductions

- **Syntax**
  
  ```
  reduce-expr:
   reduce-op reduce iterator-expr
  ```

- **Semantics**
  - Combines argument values using `reduce-op`
  - *Reduce-op* may be built-in or user-defined

- **Examples**

  ```
  total = + reduce A;
  bigDiff = max reduce [i in InnerD] abs(A(i)-B(i));
  (minVal, minLoc) = minloc reduce (A, D);
  ```
Reduction and Scan Operators

- **Built-in**
  - +, *, &&, ||, &, |, ^, min, max
  - minloc, maxloc
    - Takes a tuple of values and indices
    - Generates a tuple of the min/max value and its index

- **User-defined**
  - Defined via a class that supplies a set of methods
  - Compiler generates code that calls these methods

**Based on:**

Outline

- Domains and Arrays
- Other Domain Types
- Reductions
- NAS MG Stencil Example
Revisiting the \( rprj3 \) Stencil from NAS MG

\[
\begin{align*}
\text{left} &= \text{right} \\
\text{left} &= \text{middle} + \text{right} \\
\text{left} &= \text{middle} + \text{right} + \text{right}
\end{align*}
\]
proc rprj3(S: [?SD], R: [?RD]) {
    const Stencil = [-1..1, -1..1, -1..1],
    W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
    W3D = [(i,j,k) in Stencil] W[(i!=0) + (j!=0) + (k!=0)];

    forall ijk in SD do
        S[ijk] = + reduce [offset in Stencil]
            (W3D[offset] * R[ijk + RD.stride*offset]);
}
• Most features implemented and working correctly
• Regular and irregular domains/arrays generating parallelism
• Scalar performance lacking in some cases (particularly higher-dimensional operations)
• Implementation of unstructured domains/arrays is correct but inefficient
Future Directions

- Gain more experience with unstructured (graph-based) domains and arrays
Questions?