Chapel: Data Parallelism

Steve Deitz
Cray Inc.
Outline

- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
- Data Parallel Operations
Domains

- A first-class index set
  - Specifies size and shape of arrays
  - Supports iteration, array operations
  - Potentially distributed across machines
- Three main classes
  - Arithmetic—indices are Cartesian tuples
  - Associative—indices are hash keys
  - Opaque—indices are anonymous
- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s region concept
config const m = 4, n = 8;

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

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

- **Syntax**
  
  ```chapel
type array-type: [ domain_expr ] type
```

- **Semantics**
  
  Associates data with each index in `domain_expr`.

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

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

- For loops (discussed already)
  - Executes loop body once per loop iteration
  - Order is serial

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

- Forall loops
  - Executes loop body once per loop iteration
  - Order is parallel (must be *serializable*)

```
forall i in InnerD do ...
```
For all loops also support...

- A symbolic shorthand:

\[
[(i,j) \text{ in } D] \ A(i,j) = i + j/10.0;
\]

- An expression-based form:

\[
A = \text{forall} \ (i,j) \text{ in } D \text{ do } i + j/10.0;
\]

- A sugar for array initialization:

\[
\text{var} \ A: [(i,j) \text{ in } D] \ \text{real} = i + j/10.0;
\]
Usage of For, Forall, and Coforall

- Use for when
  - A loop must be executed serially
  - One task is sufficient for performance
- Use forall when
  - The loop can be executed in parallel
  - The loop can be executed serially
- Use coforall when
  - The loop must be executed in parallel
    (And not just for performance reasons!)
Other Domain Functionality

- Domain methods (exterior, interior, translate, ...)
- Domain slicing (intersection)
- Array slicing (sub-array references)

\[ A(\text{InnerD}) = B(\text{InnerD}); \]

- Array reallocation
  - Reassign domain → change array
  - Values are preserved (new elements initialized)

\[ D = [1..m+1, 1..m]; \]
Outline

• Domains and Arrays
• Other Domain Types
  • Strided
  • Sparse
  • Associative
  • Opaque
• Data Parallel Operations
The Varied Kinds of Domains

```chapel
var Dense: domain(2) = [1..10, 1..20],
   Strided: domain(2) = Dense by (2, 4),
   Sparse: subdomain(Dense) = genIndices(),
   Associative: domain(string) = readNames(),
   Opaque: domain(opaque);
```

**Dense**

**Strided**

**Sparse**

**Opaque**

**Associative**

<table>
<thead>
<tr>
<th>George</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
</tr>
<tr>
<td>Thomas</td>
</tr>
<tr>
<td>James</td>
</tr>
<tr>
<td>Andrew</td>
</tr>
<tr>
<td>Martin</td>
</tr>
<tr>
<td>William</td>
</tr>
</tbody>
</table>
The Varied Kinds of Arrays

```chapel
var DenseArr: [Dense] real,
StridedArr: [Strided] real,
SparseArr: [Sparse] real,
AssociativeArr: [Associative] real,
OpaqueArr: [Opaque] real;
```

- **DenseArr**: A dense array contains all elements.
- **StridedArr**: A strided array accesses elements in a non-contiguous manner.
- **SparseArr**: A sparse array stores only non-zero elements.
- **OpaqueArr**: An opaque array uses a custom representation.
- **AssociativeArr**: An associative array allows access via keys.

Diagram representations of each type are shown, illustrating their unique characteristics.
forall (i,j) in Strided {
    DenseArr(i,j) += SparseArr(i,j);
}
Associative Domains and Arrays by Example

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

Presidents += "William";

var Ages: [Presidents] int,
    Birthdays: [Presidents] string;

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

forall president in Presidents do
    if Birthdays(president) == today then
        Ages(president) += 1;
```

<table>
<thead>
<tr>
<th>Presidents</th>
<th>Ages</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>Birthdays</th>
<th>Ages</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
• Data Parallel Operations
  • Promotion
  • Reductions and scans
Functions/operators expecting scalars can also take...

- **Arrays**, causing each element to be passed
  
  \[
  \ldots \sin(A) \ldots \\
  \ldots 2*A \ldots \\
  \approx \\
  \ldots [a \text{ in } A] \sin(a) \ldots \\
  \ldots [a \text{ in } A] 2*a \ldots \\
  \]

- **Domains**, causing each index to be passed
  
  \[
  \text{foo(Sparse);} \quad \text{// calls foo for all indices in Sparse}
  \]

Multiple arguments can promote using either...

- **Zipper promotion**
  
  \[
  \ldots \text{pow}(A, B) \ldots \\
  \approx \\
  \ldots [(a, b) \text{ in } (A, B)] \text{ pow}(a, b) \ldots \\
  \]

- **Tensor promotion**
  
  \[
  \ldots \text{pow}[A, B] \ldots \\
  \approx \\
  \ldots [(a, b) \text{ in } [A,B]] \text{ pow}(a, b) \ldots \\
  \]
Reductions

• Syntax
  
  \[
  \text{reduce-expr:}
  \]
  
  \[
  \text{reduce-op reduce iterator-expr}
  \]

• Semantics
  • Combines iterated elements with \textit{reduce-op}
  • \textit{Reduce-op} may be built-in or user-defined

• Examples
  
  \[
  \text{total} = + \text{reduce } A;
  \]
  
  \[
  \text{bigDiff} = \text{max reduce} [i \text{ in InnerD}] \text{ abs}(A(i) - B(i)));
  \]
Scans

- **Syntax**
  
  \[
  \text{scan-expr:} \\
  \quad \text{scan-op scan iterator-expr}
  \]

- **Semantics**
  
  - Computes parallel prefix of \textit{scan-op} over elements
  - \textit{Scan-op} may be any \textit{reduce-op}

- **Examples**

  ```chapel
  var A, B, C: [1..5] int;
  A = 1;                   // A: 1 1 1 1 1 1
  B = + scan A;            // B: 1 2 3 4 5
  B(3) = -B(3);            // B: 1 2 -3 4 5
  C = min scan B;          // C: 1 1 -3 -3 -3
  ```
**Reduction and Scan Operators**

- **Built-in**
  - +, *, &&, | |, & , |, ^, min, max
  - minloc, maxloc
    (Generate a tuple of the min/max and its index)

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

**More information:**

Questions?

- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
  - Promotion
  - Reductions and scans