Chapel: Data Parallelism

Steve Deitz
Cray Inc.
Outline

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

```plaintext
array-type:
  [ domain-expr ] type
```

• Semantics
  • Associates data with each index in `domain-expr`

• Example

```plaintext
var A, B: [D] real;
```

• Revisited example

```plaintext
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

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

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

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

- A symbolic shorthand:

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

- An expression-based form:

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

- A sugar for array initialization:

\[
\text{var } A: [(i, j) \in D] \text{ real } = i + j/10.0;
\]
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 \(ightarrow\) 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
• Examples
The Varied Kinds of Domains

\[
\text{var Dense: } \text{domain}(2) = [1..10, 1..20], \\
\text{Strided: } \text{domain}(2) = \text{Dense by} (2, 4), \\
\text{Sparse: } \text{subdomain}(\text{Dense}) = \text{genIndices}(), \\
\text{Associative: } \text{domain(\text{string})} = \text{readNames}(), \\
\text{Opaque: } \text{domain(\text{opaque})};
\]
The Varied Kinds of Arrays

\begin{verbatim}
var DenseArr: [Dense] real,
StridedArr: [Strided] real,
SparseArr: [Sparse] real,
AssociativeArr: [Associative] real,
OpaqueArr: [Opaque] real;
\end{verbatim}
forall (i,j) in Strided {
    DenseArr(i,j) += SparseArr(i,j);
}

(Also, all domains support slicing, reallocation, ...)
**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;
```

**Presidents**
- George
- John
- Thomas
- James
- Andrew
- Martin
- William

**Birthdays**
- Feb 22
- Oct 30
- Apr 13
- Mar 16
- Mar 15
- Dec 5
- Feb 9

**Ages**
- Feb 22: 277
- Oct 30: 274
- Apr 13: 266
- Mar 16: 251
- Mar 15: 242
- Dec 5: 227
- Feb 9: 236
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

  ```
  ...\sin(A)... \\
  ...2*A...
  ```

  ≈

  ```
  ...[\text{a in A}] \ \sin(a)... \\
  ...[\text{a in A}] \ 2*a...
  ```

- **Domains**, causing each index to be passed

  ```
  \text{foo}(\text{Sparse}); \ // \ \text{calls foo for all indices in Sparse}
  ```

Multiple arguments can promote using either...

- **Zipper promotion**

  ```
  ...\text{pow}(A, B)... \\
  ```

  ≈

  ```
  ...[(a,b) \ \text{in} \ (A,B)] \ \text{pow}(a,b)... \\
  ```

- **Tensor promotion**

  ```
  ...\text{pow}[A, B]...
  ```

  ≈

  ```
  ...[(a,b) \ \text{in} \ [A,B]] \ \text{pow}(a,b)... \\
  ```
Reductions

• Syntax

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

• Semantics

• Combines iterated elements with \text{reduce-op}

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

- **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:**

Outline

- Domains and Arrays
- Other Domain Types
- Data Parallel Operations
- Examples
  - NAS MG stencil revisited
  - Array copy before key
NAS MG Stencil Revisited

= $w_0$

= $w_1$

= $w_2$

= $w_3$
def rprj3(S, R) {
    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 inds in S.domain do
        S(inds) =
            + reduce [offset in Stencil] (W3D(offset) *
            R(inds + offset*R.stride));
}
var A, B: [1..n] real, key: real;

for i in 1..n {
    if A(i) == key then
        break;
    A(i) = B(i);
}
var A, B: [1..n] real, key: real;

forall i in 1..n {
    if A(i) == key then
        break;
    A(i) = B(i);
}
```chapel
var A, B: [1..n] real, key: real;

var loc: int = n;
forall i in 1..n do
    if A(i) == key then
        loc = min(loc, i)
forall i in 1..loc do
    A(i) = B(i);
```
var A, B: [1..n] real, key: real;

var loc: sync int = n;
forall i in 1..n do
  if A(i) == key then
    loc = min(loc, i)
forall i in 1..loc do
  A(i) = B(i);
The maxloc Reduction

```chapel
var A, B: [1..n] real, key: real;

var (val, loc) =
    maxloc reduce (A==key, 1..n);
if val == false then
    loc = n;
forall i in 1..loc do
    A(i) = B(i);
```
Questions?

- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
  - Promotion
  - Reductions and scans
- Examples
  - NAS MG stencil revisited
  - Array copy before key