Chapel: Data Parallelism

Sung-Eun Choi and 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 locales
• 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**
  
  ```
  array-type:
  [ domain-expr ] elt-type
  ```

- **Semantics**
  - Stores element for each index in `domain-expr`

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

- **Revisited example**
  ```
  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 ...
```
Forall loops also support...

- A shorthand:

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

- An expression-based form:

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

- A shorthand expression-based form:

\[
A = [(i, j) \text{ in } D] i + j/10.0;
\]
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
  - Degree of concurrency $<<$ # of iterations

- **Use coforall when**
  - The loop must be executed in parallel
    (And not just for performance reasons!)
  - Each iteration has substantial work
Data Parallelism Configuration Constants

- **--dataParTasksPerLocale=#**
  - Specify # of tasks to execute forall loops
  - Default: number of cores *(in current implementation)*

- **--dataParIgnoreRunningTasks=[true | false]**
  - If false, reduce # of forall tasks by # of running tasks
  - Default: true *(in current implementation)*

- **--dataParMinGranularity=#**
  - If > 0, reduce # of forall tasks if any task has fewer iterations
  - Default: 0 *(in current implementation)*
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 \( \rightarrow \) change array
  - Values are preserved (new elements initialized)

\[ D = [1..m+1, 1..m]; \]
Array Arguments and Aliases

- Arrays are passed by reference

```chapel
def f(A: []) { A = 0; }
f(A[InnerD]);
```

- Non-argument array alias of a slice

```chapel
var AA => A(InnerD);
```

- Re-indexing arrays

```chapel
def f(A: [1..n-2,1..m-2]);
f(A[2..n-1,2..m-1]);
```

```chapel
var AA: [1..n-2,1..m-2] => A[2..n-1,2..m-1];
```
Outline

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

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

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

(Also, all domains support slicing, reallocation, ...)

All Domains Support Iteration
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;
Outline

- Domains and Arrays
- Other Domain Types
- Data Parallel Operations
  - Promotion
  - Reductions
  - Scans
- Examples
Functions/operators expecting scalars can also take...

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

- Domains, causing each index to be passed
  \[
  \text{foo}(\text{Sparse}) \approx \text{forall } i \in \text{Sparse} \text{ do } \text{foo}(i)
  \]

Multiple arguments can promote using either...

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

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

- Syntax

```
reduce-expr:
  reduce-op reduce iterator-expr
```

- Semantics
  - Combines iterated elements with `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));
```
Scans

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

- **Semantics**
  - Computes parallel prefix of `scan-op` over elements
  - *Scan-op* may be any *reduce-op*

- **Examples**
  
  ```
  var A, B, C: [1..5] int;
  A = 1;                   // A:  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
  - Pre-key copy kernel
  - \( A = B + C \)
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);
}

error: break is not allowed in forall statement
var A, B: [1..n] real, key: real;

var loc: int = n;
for i in 1..n do
  if A(i) == key {  
    loc = i-1;
    break;
  }

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-1);

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

var (val, loc) = maxloc reduce (A==key, 1..n);
if val == false then
    loc = n
else
    loc = loc - 1;

forall i in 1..loc do
    A(i) = B(i);
The Many Ways of Writing $A = B + C$

1. Whole array assignment and promotion of ‘+’

```plaintext
A = B + C;
```

2. Forall loop over arrays

```plaintext
forall (a,b,c) in (A,B,C) do
  a = b + c;
```

1. Forall loop over domain (assumes arrays’ domains equal D)

```plaintext
forall i in D do
  A(i) = B(i) + C(i);
```

2. Forall loop over arrays’ domains

```plaintext
forall (i,j,k) in (A.domain, B.domain, C.domain) do
  A(i) = B(j) + C(k);
```
5. Whole array assignment and promotion of ‘+’ of slices
   \[ A(D) = B(D) + C(D); \]

6. Forall loop over array slices
   \[
   \text{forall} \ (a, b, c) \text{ in } (A(D), B(D), C(D)) \text{ do } \\
   \hspace{1cm} a = b + c;
   \]

7. Whole array assignment and promotion of slices with ranges
   \[
   A[.., D\.low(2)..D\.high(2)] = \hspace{1cm} B[..D\.high(1), D\.low(2)..] + C[D\.dims(1), ..];
   \]

8. Forall loop over A’s domain with re-indexed aliases of B and C
   \[
   \text{var} \ BB: [A\.domain] \Rightarrow B, \ CC: [A\.domain] \Rightarrow C; \hspace{1cm} [i \text{ in } A\.domain] \ A(i) = BB(i) + CC(i);
   \]
The Many Ways of Writing $A = B + C$

9. Whole array assignment of a forall expressions

$$A = \text{forall } (b, c) \text{ in } (B, C) \text{ do } b + c;$$

10. Forall loop over an array and the promotion of ‘+’

$$\text{forall } (a, bc) \text{ in } (A, B+C) \text{ do }$$
$$a = bc;$$

11. Assignment to a forall expression from promoted expressions

$$([a \text{ in } A] a) =$$
$$([b \text{ in } B] b) + [c \text{ in } C] c;$$

12. Forall loop over arrays and a forall expression

$$\text{forall } (a, b, c) \text{ in } (A, B, \text{forall } c \text{ in } C \text{ do } c) \text{ do }$$
$$a = b + c;$$
13. Task-parallel loop with data-parallel array slice assignments

\[
\text{coforall } t \text{ in } 1..\text{numTasks} \text{ do }
\]

\[
A(\text{chunk}(D,t)) = B(\text{chunk}(D,t)) + C(\text{chunk}(D,t));
\]

- Similar performance is ultimately expected for all variations
- Compiler translates code using array implementations
  - All arrays are implemented in Chapel (Distributions and Layouts)
  - Compiler targets a structural interface in Chapel

Note: These examples are overkill for \( A=B+C \), but they demonstrate the orthogonality and power of Chapel’s data-parallel abstractions.
Promotion of ‘+’ produces a `forall` expression

\[
A = \text{forall} \ (b, c) \in (B, C) \text{ do } b+c;
\]

Array assignment is implemented via a zippered `forall` loop

\[
\text{forall} \ (a, (b, c)) \in (A, \text{forall} \ (b, c) \in (B, C) \text{ do } b+c) \text{ do}
\]

\[
a = b + c;
\]

Parallel iteration over a default array uses a `coforall`

\[
\text{coforall} \ t \text{ in } 1..\text{numTasks} \text{ do}
\]

\[
\text{forall} \ (a, (b, c)) \in (A(\text{chunk}(D, t)),
(\text{forall} \ (b, c) \in (B(\text{chunk}(D, t)), C(\text{chunk}(D, t)))) \text{ do }
\]

\[
a = b + c;
\]
Questions?

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

- Examples
  - NAS MG stencil revisited
  - Pre-key copy kernel
  - \( A = B + C \)