Chapel: Data Parallelism
Outline

- Domains and Arrays
  - Regular Domains and Arrays
  - Iterations and Operations
- Other Domain Types
- Reductions and Scans
- NAS MG Stencil Revisited
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;

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**
  
  ```
  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)**
  - Execute loop body once per domain index, serially
  - Index variable takes on const index values

```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 domain index values (*const*)

Chapel: Data Parallelism
For all loops also support...

- A shorthand:

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

- Expression-based forms:

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

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

- **Methods for creating new domains**
  ```chapel
  var D2 = InnerD.expand(1,0);
  ```

- **Overloaded Operators**
  ```chapel
  var D3 = InnerD + (0,1);
  ```

- **Intersection via Slicing**
  ```chapel
  var D4 = D2[D3];
  ```
Array Slicing/Sub-Arrays

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

\[ A[\text{InnerD}] = B[\text{InnerD} + (0,1)]; \]
Array Reallocation

Reassigning a domain logically reallocates its arrays
- values are preserved for common indices

\[ D = [1..2*m, 1..2*n]; \]
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;
```
Array Arguments and Aliases

- Array values are passed by reference
  
  ```chapel
def zero(X: []) { X = 0; }
  
  zero(A[InnerD]); // zeroes the inner values of A
  ```

- Formal array arguments can reindex actuals
  
  ```chapel
def f(X: [1..b,1..b]) { ... } // X uses 1-based indices
  
f(A[lo..#b, lo..#b]);
  ```

- Array alias declarations provide similar functionality
  
  ```chapel
  var InnerA => A[InnerD];
  var InnerA1: [1..n-2,1..m-2] => A[2..n-1,2..m-1];
  ```
Promoted Functions and Operators

Functions/operators expecting scalars can also take...

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

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

Multiple arguments can promote using either...

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

- **Tensor product promotion**
  
  \[
  \text{pow}[A, B] \quad \approx \quad \text{forall } (a,b) \text{ in } [A,B] \text{ do } \text{pow}(a,b)
  \]
By default*, controlled by three configuration variables:

--dataParTasksPerLocale=#
  • Specify # of tasks to execute forall loops
  • Current Default: number of cores

--dataParIgnoreRunningTasks=[true | false]
  • If false, reduce # of forall tasks by # of running tasks
  • Current Default: true

--dataParMinGranularity=#
  • If > 0, reduce # of forall tasks if any task has fewer iterations
  • Current Default: 1

*Default values can be overridden by domain map arguments
Outline

- Domains and Arrays
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Reductions and Scans
- 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();
```

- **dense**
- **strided**
- **sparse**

```chapel
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;

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;

Chapel: Data Parallelism
Slicing

...to slice arrays...

Ocean[AirSpace] += IceCaps[AirSpace];

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

"steve"
"lee"
"sung"
"david"
"jacob"
"albert"
"brad"
Reallocation

...and to reallocate arrays

\[
\text{AirSpace} = \text{OceanSpace} \text{ by } (2,2);
\]
\[
\text{IceSpace} += \text{genEquator}();
\]

newnode = Vertices.create();  People += “srini”;
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;
Outline

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

- **Syntax**

  \[
  \text{reduce-expr:}
  \begin{align*}
  & \text{reduce-op reduce iterator-expr}
  \end{align*}
  \]

- **Semantics**
  - Combines argument values using \textit{reduce-op}
  - \textit{Reduce-op} may be built-in or user-defined

- **Examples**

  \[
  \begin{align*}
  \text{total} &= + \text{reduce } A; \\
  \text{bigDiff} &= \text{max reduce } [i \ \text{in InnerD}] \text{ abs}(A(i)-B(i)); \\
  (\text{minVal, minLoc}) &= \text{minloc reduce } (A, D);
  \end{align*}
  \]
Scans

- **Syntax**

```
scan-expr:
  scan-op scan iterator-expr
```

- **Semantics**
  - Computes parallel prefix over values using `scan-op`
  - *Scan-op* may be any *reduce-op*

- **Examples**

```chapel
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`
    - 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 and Scans
- NAS MG Stencil Revisited
Revisiting the \textit{rprj3} Stencil from NAS MG

\begin{align*}
\text{Cube 1} & \quad \text{= } \quad \text{Cube 2} \\
\text{Cube 2} & \quad \text{= } \quad \text{Cube 3} + \text{Cube 4} + \text{Cube 5}
\end{align*}
def 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]);
}
Data Parallelism: Status

- Most features implemented and working correctly
- Regular domains/arrays generating parallelism
- Irregular domain/array operations currently serialized
- Scalar performance lacking for higher-dimensional domain/array operations
Future Directions

- Fix lacks on previous slides
- Gain more experience with graph-based domains/arrays
Questions?

- Domains and Arrays
  - Regular Domains and Arrays
  - Iterations and Operations
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
  - Reductions
  - Scans
- NAS MG stencil revisited