Outline

- Domains and Arrays
  - overview
  - arithmetic domains
  - domain roles
- Other Domain Types
- Data Parallel Operations
- Example Computations
Domains

- **domain**: a first-class index set
  - specifies size and shape of arrays
  - supports iteration, array operations
  - potentially distributed across machine

- Three main classes:
  - **arithmetic**: indices are Cartesian tuples
    - rectilinear, multidimensional
    - optionally strided and/or sparse
  - **associative**: indices serve as hash keys
    - supports hash tables, dictionaries
  - **opaque**: indices are anonymous
    - supports sets, graph-based computations

- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s region concept

Sample Arithmetic Domains

```
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
```
Sample Arithmetic Domains

```plaintext
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
var InnerD: subdomain(D) = [2..m-1, 2..n-1];
```

Domain Roles: Declaring Arrays

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

- **Semantics**
  - for each index in `domain-expr`, stores an element of `type`

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

- **Revisiting our previous array declarations:**
  ```plaintext
  var A: [1..3] int; // creates an anonymous domain [1..3]
  ```
Domain Roles: Supporting Iteration

- **Syntax**
  ```chapel
  for index-expr in domain-expr loop-body
  forall index-expr in domain-expr loop-body
  ```

- **Semantics**
  - *for* – same as previous for-loops we’ve seen; indices are const
  - *forall* – asserts the loop iterations can/should be executed in parallel
    – also that they are serializable (can be run by a single task)

- **Example**
  ```chapel
  forall (i,j) in InnerD do
    A(i,j) = i + j/10.0;
  for ind in InnerD { write(A(ind), " "); }
  ```

- **Output**
  ```plaintext
  2.2 2.3 2.4 2.5 2.6 2.7 3.2 3.3 3.4 3.5 3.6 3.7
  ```

Other forall loop forms

- Forall loops also support...
  ...
  ...an expression-based form:
  ```chapel
  var A: [D] real = forall (i,j) in D do i + j/10.0;
  ```

  ...
  ...a symbolic shorthand:
  ```chapel
  [(i,j) in D] A(i,j) = i + j/10.0;
  ```

  ...
  ...and a sugar that combines it with the array type declaration syntax:
  ```chapel
  var A: [D] real = [(i,j) in D] i + j/10.0;
  ```
  ```plaintext
  // can be written:
  ```
  var A: [(i,j) in D] real = i + j/10.0;
  ```
Loops and Parallelism

- **for loops**: one task executes all iterations

- **forall loops**: some number of tasks executes the iterations
  • as determined by the iterator expression controlling the loop
    • for domains/arrays, specified as part of its *distribution*
    • for other objects/iterators, author specifies using task parallelism

- **coforall loops**: one task per iteration

Domain Roles: Domain Slicing

- **Syntax**
  
  ```chapel
domain-slice:
  domain-expr[domain-expr]
  ```

- **Semantics**
  • evaluates to the intersection of the two domains

- **Example**
  ```chapel
const SubD: subdomain(D) = D[2..,..7];
  ```
Domain Roles: Array Slicing

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

- **Semantics**
  
  - evaluates to the sub-array referenced by \textit{domain-expr}
  - \textit{domain-expr}'s indices must be legal for \textit{array-expr}

- **Example**
  
  
  \[
  A[\text{InnerD}] = B[\text{InnerD}];
  \]

Domain Roles: Array Reallocation

- **Semantics**
  
  - re-assigning a domain's index set causes its arrays to be reallocated
  - array values are preserved for indices that remain in the index set
  - elements for new indices are initialized to the type’s default value

- **Example**
  
  
  \[
  D = [1..2^m, 1..2^n];
  \]
Outline

- Domains
- Other Domain Types
  - strided
  - sparse
  - associative
  - opaque
- Data Parallel Operations
- Example Computations

Other Domain Types

Domain indices can be…

...integer tuples…

...anonymous…...or arbitrary values.

dense
(1,0)
strided
(10,24)
sparse
(10,24)
graphs

"George"
"John"
"Thomas"
"James"
"Andrew"
"Martin"
"William"
Domain Declarations

\[
\begin{align*}
\text{var } & \text{ DnsDom: domain(2) = [1..10, 0..24],} \\
& \text{StrDom: subdomain(DnsDom) = DnsDom by (2,4),} \\
& \text{SpsDom: subdomain(DnsDom) = genIndices();}
\end{align*}
\]

Array Declarations

\[
\begin{align*}
\text{var } & \text{ DnsArr: [DnsDom] complex,} \\
& \text{StrArr: [StrDom] real(32),} \\
& \text{SpsArr: [SpsDom] real;}
\end{align*}
\]
**Data Parallelism: Domain Iteration**

All domain types support iteration...

```plaintext
forall ij in StrDom {
    DnsArr(ij) += SpsArr(ij);
}
```

**Data Parallelism: Array Slicing**

...array slicing...

```plaintext
DnsArr[StrDom] += SpsArr[StrDom];
```
Data Parallelism: Array Reallocation

...array reallocation (and all other domain/array operations)

StrDom = DnsDom by (2, 2);
SpsDom += genEquator();

The Domain/Index Hierarchy

forall ij in InnerD { ...A(ij)... }

ij implicitly defined as:
const ij: index(InnerD);

No bounds check needed since
index(InnerD) ∈ InnerD ⊂ D => domain(A)
### Associative Domains and Arrays

```chapel
var People: domain(string);
var Age: [People] int,
    Birthdate: [People] string;

People += "john";
Age("john") = 60;
Birthdate("john") = "12/11/1943";

... 
forall person in People {
    if (Birthdate(person) == today) {
        Age(person) += 1;
    }
}
```

### Opaque Domains and Arrays

```chapel
var Vertices: domain(opaque);

for i in (1..5) {
    Vertices.create();
}

var AV, BV: [Vertices] real;
```
Opaque Domains and Arrays

```chapel
define Vertices
class domain(opaque)
end

var left, right: [Vertices] index(Vertices);
var root: index(Vertices);
root = Vertices.create();
left(root) = Vertices.create();
right(root) = Vertices.create();
left(right(root)) = Vertices.create();
```

**conceptually:**

```
root
```

**more precisely:**

```
Vertices
```

Outline

- Domains
- Other Domain Types
- Data Parallel Operations
  - promotion
  - reductions and scans
- Example Computations
Promotion

- Functions/operators expecting scalar values can also take...
  ...arrays, causing each element to be passed in
  
  ```
  A = sin(B);
  B = 2 * A;
  ```

  ...domains, causing each index to be passed in
  
  ```
  def foo(x: (int, int)) { ... }
  foo(SpsDom);  // calls foo once per index in SpsDom
  ```

- When multiple arguments are promoted, calls may use...
  ...zippered promotion:
  
  ```
  X = pow(A, B);  // X is 2D; X(i,j) = pow(A(i,j), B(i,j))
  ```

  ...tensor promotion:
  
  ```
  Y = pow[A, B];  // Y is 2x2D;
  // Y(i,j)(k,l) = pow(A(i,j), B(k,l))
  ```

Promotion and Parallelism

- Promoted functions/operators are executed in parallel
  - as if a forall loop implements the calls using zipper/tensor iteration

  ```
  A = sin(B);
  B = 2 * A;
  foo(SpsDom);
  X = pow(A, B);
  ```

  ```
  A = [b in B] sin(b);
  B = [a in A] 2 * a;
  [i in SpsDom] foo(i);
  X = [(a,b) in (A,B)] pow(a,b);
  ```
Reductions

- Syntax
  
  \[ \text{reduce-expr:} \]
  \[ \text{reduce-type} \ \text{reduce} \ \text{iteratable-expr} \]

- Semantics
  
  - combines elements generated by \text{iteratable-expr} using \text{reduce-type}
  - \text{reduce-type} may be one of several built-in operators, or user-defined

- Examples
  
  \[ \text{tot} = + \ \text{reduce} \ A; \quad \text{// tot is the sum of all elements in A} \]
  \[ \text{big} = \text{max} \ \text{reduce} \ [i \ \text{in} \ \text{InnerD}] \ \text{abs}(A(i) + B(i)); \]

- Future work:
  
  - support for partial reductions to reduce only a subset of an array's dimensions

Scans

- Syntax
  
  \[ \text{scan-expr:} \]
  \[ \text{scan-type} \ \text{scan} \ \text{iteratable-expr} \]

- Semantics
  
  - combines elements generated by \text{iteratable-expr} using \text{scan-type}, generating partial results along the way
  - \text{scan-type} may be one of several built-in operators, or user-defined

- Examples
  
  \[ \text{var} \ A, B, C: [1..5] \ \text{real}; \]
  \[ A = 1.1; \quad \text{// A is: 1.1 1.1 1.1 1.1 1.1} \]
  \[ B = + \ \text{scan} \ A; \quad \text{// B is: 1.1 2.2 3.3 4.4 5.5} \]
  \[ B(3) = -B(3); \quad \text{// B is: 1.1 2.2 -3.3 4.4 5.5} \]
  \[ C = \text{min} \ \text{scan} \ B; \quad \text{// C is: 1.1 1.1 -3.3 -3.3 -3.3} \]
Reduction/Scan operators

- Built-in:
  - +, -, *, /, &, ^, &, |, min, max: do the obvious things
  - minloc, maxloc: generate a tuple result: (min/max value, its index)

- User-defined:
  - user must define a class that supports a number of methods to:
    - generate a new identity state value
    - combine the state element with a new element
    - combine two state elements
    - generate an output result
    - ...
  - the compiler generates a code template to compute the operation in parallel, utilizing the user’s class methods
  - for more information, see:

Outline

- Domains
- Other Domain Types
- Data Parallel Operations
- Example Computations
  - Jacobi iteration
  - Multigrid
Jacobi Iteration in Pictures

\[
\sum \left( \begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\end{array} \right) \div 4
\]

repeat until max change < \( \varepsilon \)

\[A: \begin{array}{|c|c|c|c|c|}
\hline
\multicolumn{5}{|c|}{n} \\
\hline
\multicolumn{5}{|c|}{1.0} \\
\hline
\end{array}
\]

Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);
var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    [((i,j)) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;
    var delta = max reduce abs(A(D) - Temp(D));
    A[D] = Temp[D];
}
while (delta > epsilon);
writeln(A);
```

**Declare program arguments**
- `const` can’t change values after initialization
- `config` can be set on executable command-line

Command example:
```
prompt> jacobi --n=10000 --epsilon=0.0001
```

Note that no types are given; inferred from initializer:
- `n` ➞ integer (current default, 32 bits)
- `epsilon` ➞ floating-point (current default, 64 bits)

**Declare domains (first class index sets)**
- `domain(2)` ➞ 2D arithmetic domain, indices are integer 2-tuples
- `subdomain(P)` ➞ a domain of the same type as `P` whose indices are guaranteed to be a subset of `P`'s

**exterior** ➞ one of several built-in domain generators

Chapel: Data Parallelism (33)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [i,j in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4;
    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
} while (delta > epsilon);

writeln(A);
```

Declare arrays

- `var` can be modified throughout its lifetime
- `T` declares variable to be of type `T`
- `[D] T` array of size `D` with elements of type `T`
- *(no initializer)* values initialized to default value (0.0 for reals)

Notes:
- `indexing by domain` ⇒ slicing mechanism
- `array expressions` ⇒ parallel evaluation

Set Explicit Boundary Condition

- `indexing by domain` ⇒ slicing mechanism
Jacobi Iteration in Chapel

Compute 5-point stencil

\[(i,j) \text{ in } D \Rightarrow \text{parallel forall expression over } D's \text{ indices, binding them to new variables } i \text{ and } j\]

**Note:** since \((i,j) \subseteq D \text{ and } D \subseteq \text{BigD} \text{ and } \text{Temp}[\text{BigD}]

\Rightarrow \text{no bounds check required for } \text{Temp}(i,j)

with compiler analysis, same can be proven for A's accesses

\[
\sum \left( \frac{\text{Temp}(i,j)}{4} \right)
\]

\[
\begin{align*}
[(i,j) \text{ in } D] & \quad \text{Temp}(i,j) = (A(i-1,j) + A(i+1,j) \\
& \quad + A(i,j-1) + A(i,j+1)) / 4;
\end{align*}
\]

const delta = \text{max reduce abs}(A[D] - Temp[D]);
A[D] = Temp[D];
while (delta > epsilon);
writeln(A);

Compute maximum change

\text{op reduce} \Rightarrow \text{collapse aggregate expression to scalar using op}

**Promotion:** abs() and – are scalar operators, automatically promoted to work with array operands

\[
\begin{align*}
do & \\
& \quad [(i,j) \text{ in } D] \quad \text{Temp}(i,j) = (A(i-1,j) + A(i+1,j) \\
& \quad + A(i,j-1) + A(i,j+1)) / 4;
\end{align*}
\]

const delta = \text{max reduce abs}(A[D] - Temp[D]);
A[D] = Temp[D];
while (delta > epsilon);
writeln(A);
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
        + A(i,j-1) + A(i,j+1)) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

### Copy data back & Repeat until done

Uses slicing and whole array assignment

Standard `do...while` loop construct

Write array to console

If written to a file, parallel I/O would be used
Pelitoe Iteration in Chapel

\begin{verbatim}
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    \((i,j) in D\) Temp(i,j) = (A(i-1,j) + A(i+1,j)
                        + A(i,j-1) + A(i,j+1)) / 4;
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
\end{verbatim}
Multigrid Example

V: input array

U:

R:

Hierarchical Arrays

conceptually:

dense indexing:

strided indexing:

level 0

level 1

level 2

level 3

(1:8,1:8)

(1:4,1:4)

(1:2,1:2)

(1:1,1:1)

(1:8:1,1:8:1)

(1:8:2,1:8:2)

(1:8:4,1:8:4)

(1:8:8,1:8:8)
Hierarchical Arrays

conceptually:

dense indexing:

strided indexing:

Hierarchical Array Declarations in Chapel

```chapel
config const n = 1024,
    numLevels = lg2(n);

const Levels = [0..#numLevels];
const ProblemSpace: domain(3) distributed BlockWrap
    = [1..n, 1..n, 1..n];

var V: [ProblemSpace] real;

const HierSpace: [lvl in Levels] subdomain(ProblemSpace)
    = ProblemSpace by -2**lvl;

var U, R: [lvl in Levels] [HierSpace(lvl)] real;
```
Overview of NAS MG

MG’s projection/interpolation cycle
NAS MG: \textit{rprj3} stencil

\textbf{Multigrid: Stencils in Chapel}

- Can write them out explicitly, as in Jacobi...

\begin{verbatim}
def rprj3(S, R) {
    const w: \{0..3\} real = (0.5, 0.25, 0.125, 0.0625);
    const Rstr = R.stride;
    forall ijk in S.domain do
        S(ijk) = w(0) * R(ijk)
            + w(1) * (R(ijk+Rstr*(1,0,0)) + R(ijk+Rstr*(-1,0,0))
                        + R(ijk+Rstr*(0,1,0)) + R(ijk+Rstr*(-0,1,0))
                        + R(ijk+Rstr*(0,0,1)) + R(ijk+Rstr*(0,0,-1)));
        S(ijk) = w(2) * (R(ijk+Rstr*(1,1,0)) + R(ijk+Rstr*(1,-1,0))
                        + R(ijk+Rstr*(-1,1,0)) + R(ijk+Rstr*(-1,-1,0))
                        + R(ijk+Rstr*(1,0,1)) + R(ijk+Rstr*(1,0,-1))
                        + R(ijk+Rstr*(-1,0,1)) + R(ijk+Rstr*(-1,0,-1))
                        + R(ijk+Rstr*(0,1,1)) + R(ijk+Rstr*(0,1,-1))
                        + R(ijk+Rstr*(0,0,1)) + R(ijk+Rstr*(0,0,-1)));
        S(ijk) = w(3) * (R(ijk+Rstr*(1,1,1)) + R(ijk+Rstr*(1,1,-1))
                        + R(ijk+Rstr*(1,-1,1)) + R(ijk+Rstr*(1,-1,-1))
                        + R(ijk+Rstr*(-1,1,1)) + R(ijk+Rstr*(-1,1,-1))
                        + R(ijk+Rstr*(-1,-1,1)) + R(ijk+Rstr*(-1,-1,-1)));
}
\end{verbatim}
Multigrid: Stencils in Chapel

- or, note that a stencil is simply a reduction over a small subarray expression
- Thus, stencils can be written in a “syntactically scalable” way using reductions:

```chapel
def rprj3(S, R) {
    const Stencil: domain(3) = [-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 S.domain do
        S(ijk) = + reduce [off in Stencil]
            (w3d(off) * R{ijk + R.stride*off});
}
```

NAS MG rprj3 stencil in Fortran+MPI
Data Parallelism Status

- **Stable Features:**
  - most features in this section are implemented, but using a single task

- **Incomplete Features:**
  - forall loops, promotion, and reductions do not result in parallelism yet
  - promoted functions do not preserve array shape by default
  - index types and subdomains are not bounds-checked

- **Unimplemented Features:**
  - arrays of differently-sized arrays are not yet supported
  - partial reductions and scans are not yet defined or implemented
  - user defined reductions and scans are not yet supported

---

**Questions?**