Chapel: Data Parallelism
Data Parallelism:
- parallelism is driven by collections of data
  - data aggregates (arrays)
  - sets of indices (ranges, domains)
  - other user-defined collections
- e.g., “for all elements in array A …”

Task Parallelism:
- parallelism is expressed in terms of distinct computations
- e.g., “create a task to do foo() while another does bar()”

(Of course, data parallelism is executed using tasks and task parallelism typically operates on data, so the line can get fuzzy at times...)
"Hello World" in Chapel: a Data Parallel Version

- Data Parallel Hello World

```chapel
config const numIters = 100000;

forall i in 1..numIters do
    writeln("Hello, world! ",
            "from iteration ", i,
            " of ", numIters);
```
Outline

- Domains and Arrays
  - Rectangular Domains and Arrays
  - Iterations and Operations
- Other Domain Types
- Reductions and Scans
- Jacobi Iteration Example
Domains

**Domain:** A first-class index set

- A 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 \( D: \text{domain}(2) = [1..m, 1..n]; \)
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**
  
  ```markdown
  array-type:
  [ domain-expr ] elt-type
  ```

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

- **Example**

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

- **Earlier example, revisited**

  ```markdown
  var A: [1..3, 1..5] real; // [1..3, 1..5] is an anonymous domain
  ```
Domain Iteration

- For loops (discussed already)
  - Execute loop body once per domain index, serially

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

- Forall loops
  - Executes loop body once per domain index, in parallel
  - Loop must be *serializable* (executable by one task)

```python
forall i in InnerD do ...
```

- Loop variables take on `const` domain index values
Other Forall Loops

Forall loops also support...

- A shorthand notation:

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

- Expression-based forms:

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

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

- **Methods for creating new domains**
  
  ```javascript
  var D2 = InnerD.expand(1, 0);
  
  var D3 = InnerD.translate(0, 1);
  
  var D4 = D2[D3];
  ```

- **Intersection via Slicing**
  
  ```javascript
  var D4 = D2[D3];
  ```

- **Range operators (e.g., #, by)**
Indexing into arrays with domain values results in a sub-array expression (an “array slice”)

\[ A[\text{InnerD}] = B[\text{InnerD}.\text{translate}(0,1)] ; \]
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

```
for a in A[InnerD] do ...
```

```
forall a in A[InnerD] do ...
```

- Array loop indices refer to array variables (modifiable)

```
forall (a, (i,j)) in (A, D) do a = i + j/10.0;
```

Note that forall loops support zippered iteration, like for-loops.
Arrays can be indexed using variables of their domain’s index type (e.g., tuples) or lists of integers

```java
var i = 1, j = 2;
var ij = (i, j);
A[ij] = 1.0;
A[i, j] = 2.0;
```

Array indexing can use either parentheses or brackets

```java
A(ij) = 3.0;
A(i, j) = 4.0;
```
Arrays are passed by reference by default

```chapel
proc zero(X: []) { X = 0; }
zero(A[InnerD]);  // zeroes the inner values of A
```

Formal array arguments can reindex actuals

```chapel
proc 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];
```
Functions/operators expecting scalars can also take... ...arrays, causing each element to be passed in

\[
\begin{align*}
sin(A) & \approx \text{forall } a \in A \text{ do } \sin(a) \\
2*A & \approx \text{forall } a \in A \text{ do } 2*a
\end{align*}
\]

...domains, causing each index to be passed in

\[
\begin{align*}
\text{foo(Sparse)} & \approx \text{forall } i \in \text{Sparse} \text{ do } \text{foo}(i)
\end{align*}
\]

Multiple arguments promote using zipper promotion

\[
\begin{align*}
\text{pow(A, B)} & \approx \text{forall } (a,b) \in (A,B) \text{ do } \text{pow}(a,b)
\end{align*}
\]
forall loops are implemented using multiple tasks
  - details depend on what is being iterated over
so are operations that are equivalent to forall loops
  - promoted operators/functions, whole array assignment, ...
many times, this parallelism can seem invisible
  - for this reason, Chapel’s data parallelism can be considered *implicitly parallel*
  - it also tends to make the data parallel features easier to use and less likely to result in bugs as compared to explicit tasks
By default*, controlled by three configuration variables:

--- **dataParTasksPerLocale=*/
- Specify # of tasks to execute forall loops
- *Current Default*: number of processor 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
- Jacobi Iteration Example
Chapel Domain Types

Chapel supports several domain types...

```plaintext
var OceanSpace = [0..#lat, 0..#long],
AirSpace = OceanSpace by (2,4),
IceSpace: sparse subdomain(OceanSpace) = genCaps();
```

```
var Vertices: domain(opaque) = ..., People: domain(string) = ...;
```
All domain types can be used to declare arrays...

```chapel
def var Ocean: [OceanSpace] real,
        Air: [AirSpace] real,
        IceCaps[IceSpace] real;
```

```chapel
def var Weight: [Vertices] real,
        Age: [People] int;
```
...to iterate over index sets...

```plaintext
forall ij in AirSpace do
  Ocean[ij] += IceCaps[ij];
```

```plaintext
forall v in Vertices do
  Weight[v] = numEdges[v];
```

```plaintext
forall p in People do
  Age[p] += 1;
```
...to slice arrays...

Ocean[AirSpace] += IceCaps[AirSpace];

...Vertices[Interior]...

...People[Interns]...

“steve”
“lee”
“sung”
“david”
“jacob”
“albert”
“brad”
...and to reallocate arrays

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

```plaintext
newnode = Vertices.create();
People += "vass";
```
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
- Jacobi Iteration Example
Reductions

• **Syntax**

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

• **Semantics**
  • Combines argument values using `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]);
(minVal, minLoc) = minloc reduce (A, D);
```
Scans

• Syntax

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

• Semantics

• Computes parallel prefix over values using \textit{scan-op}
• \textit{Scan-op} may be any \textit{reduce-op}

• Examples

\begin{verbatim}
var A, B, C: [1..5] int;
A = 1;                   // A:  1  1  1  1  1
B = + scan A;            // B:  1  2  3  4  5
C = min scan B;          // C:  1  1 -3 -3 -3
\end{verbatim}
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
- Jacobi Iteration Example
Jacobi Iteration in Pictures

\[ \sum \left( \begin{array}{ccc} 1.0 \\ \end{array} \right) \div 4 \quad \text{repeat until max change} < \varepsilon \]
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;
    const delta = max reduce abs(A(D) - Temp(D));
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Declare program parameters

- **config**: can be set on executable command-line
  
  ```prompt>```
  jacobi --n=10000 --epsilon=0.0001

- **const**: can’t change values after initialization

- **note that no types are given; inferred from initializer**
  
  - `n` ⇒ default integer (32 bits)
  - `epsilon` ⇒ default real floating-point (64 bits)
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);
```

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
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;
```

**Declare arrays**

- `var` can be modified throughout its lifetime
- `: [BigD] T` array of size `BigD` with elements of type `T`  
  *no initializer*  
- `⇒` values initialized to default value (0.0 for reals)
Jacobi Iteration in 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;

Set Explicit Boundary Condition

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation
```
Jacobi Iteration in Chapel

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

Compute 5-point stencil

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

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

\[
\]

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

writeln(A);
```

Jacobi Iteration in Chapel

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

const BigD: domain(2) = [0..n+1, 0..n+1],

Compute maximum change

op reduce ⇒ collapse aggregate expression to scalar using op

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

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;
    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
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);
```

Write array to console
Jacobi Iteration in Chapel

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

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

var A, Temp : [BigD] real;
```

With this change, same code runs in a distributed manner
Domain distribution maps indices to *locales*
⇒ decomposition of arrays & default mapping of iterations to locales
Subdomains inherit parent domain’s distribution
Jacobi Iteration in Chapel

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

const BigD = [0..n+1, 0..n+1] dmapped 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);
```
Data Parallelism: Status

- Most features implemented and working correctly
- Scalar performance not optimal for higher-dimensional domain/array operations
- Implementation of unstructured domains/arrays is correct but inefficient
Future Directions

- Gain more experience with unstructured (graph-based) domains and arrays
• Domains and Arrays
  • Regular Domains and Arrays
  • Iterations and Operations
• Other Domain Types
  • Strided
  • Sparse
  • Associative
  • Opaque
• Data Parallel Operations
  • Reductions
  • Scans
• Jacobi Iteration Example

Questions?