Chapel: Data Parallelism

Data vs. Task Parallelism (Our Definitions)

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

```
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

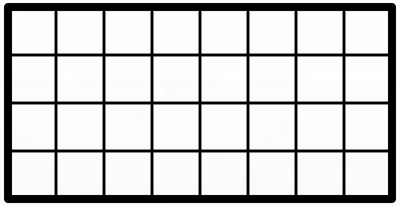
Domains

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

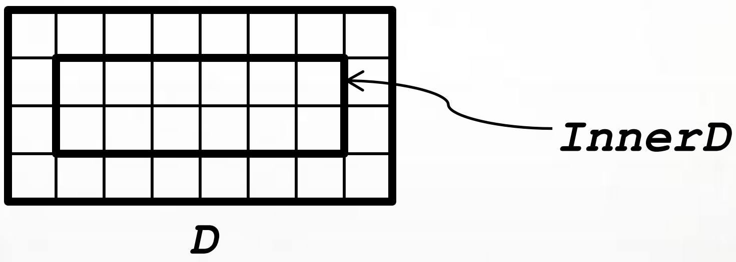
Sample Domains

```
config const m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];
```



Sample Domains

```
config const m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];

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



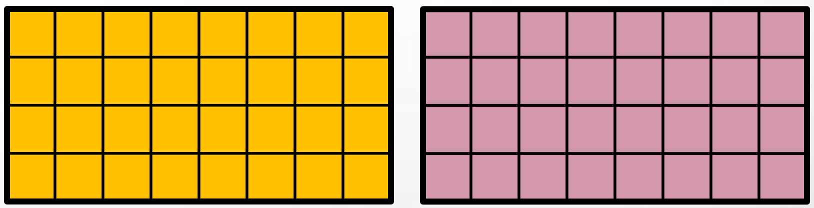
Domains Define Arrays

- Syntax

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

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

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



A *B*

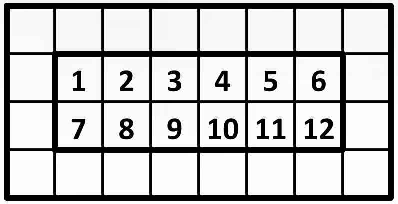
- Earlier example, revisited

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

Domain Iteration

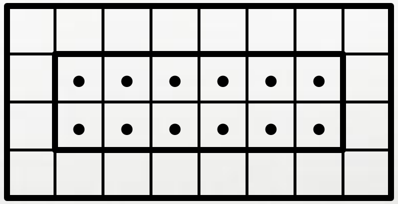
- For loops (discussed already)
  - Execute loop body once per domain index, serially

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



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

```
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) in D] A[i,j] = i + j/10.0;
```

- Expression-based forms:

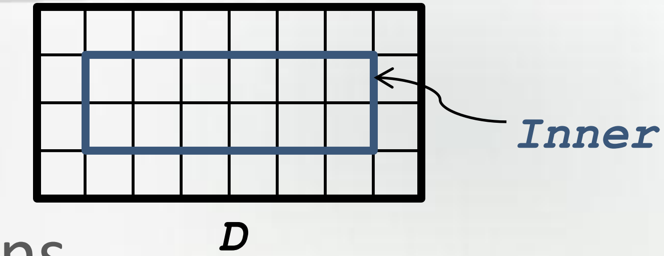
```
A = forall (i,j) in D do i + j/10.0;
```

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

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

A

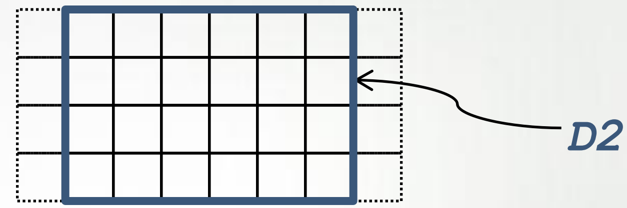
Domain Algebra



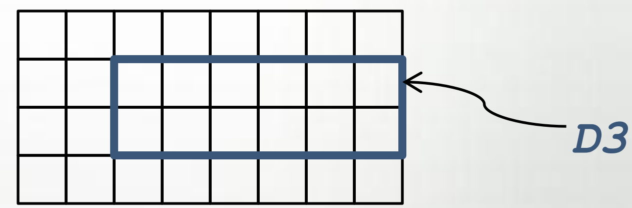
Domain values support...

- Methods for creating new domains

```
var D2 = InnerD.expand(1,0);
```

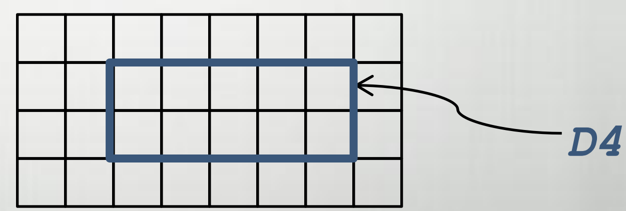



```
var D3 = InnerD.translate(0,1);
```



- Intersection via Slicing

```
var D4 = D2[D3];
```

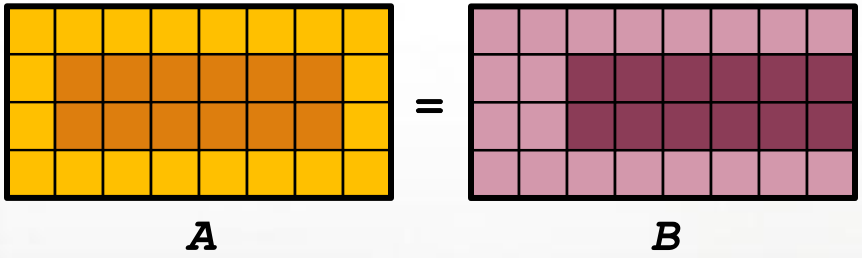


- Range operators (e.g., **`#`**, **`by`**)

Sub-Arrays/Array Slicing

Indexing into arrays with domain values results in a sub-array expression (an “array slice”)

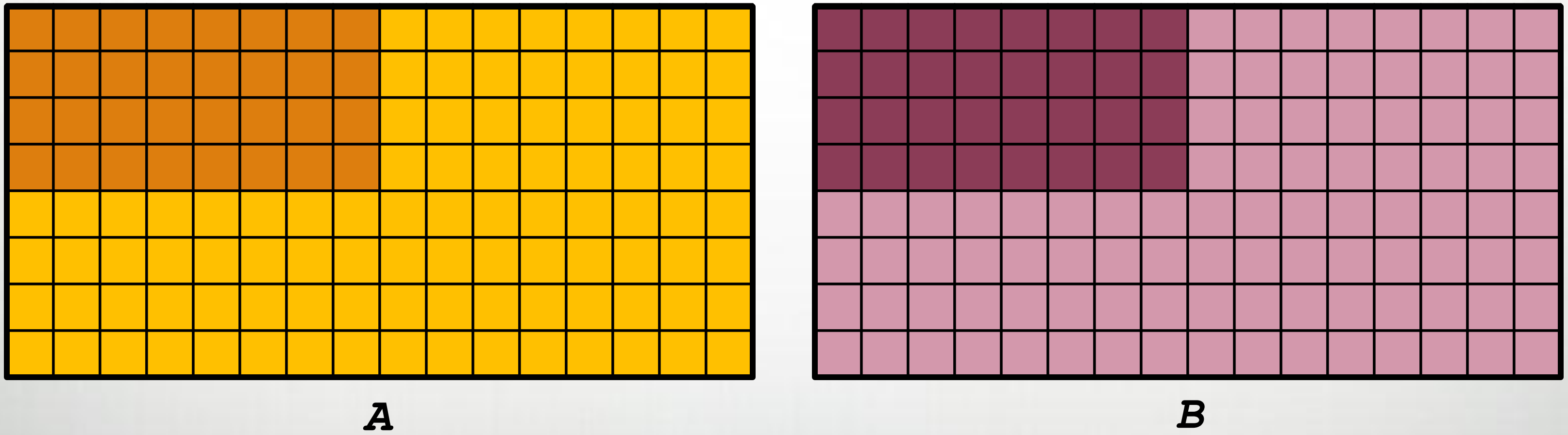
```
A[InnerD] = B[InnerD.translate(0,1)];
```



Array Reallocation

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

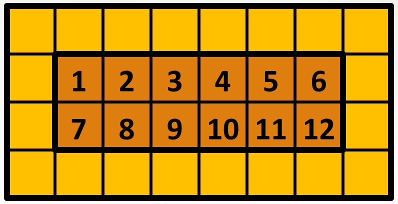
```
D = [1..2*m, 1..2*n];
```



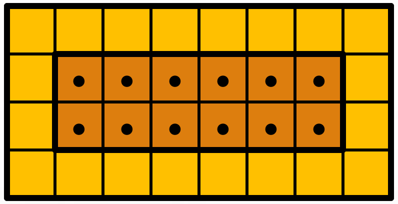
Array Iteration

- Array expressions also support for and forall loops

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



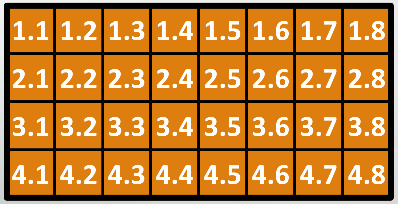

```
forall a in A[InnerD] do ...
```



- 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



Array Indexing

- Arrays can be indexed using variables of their domain's index type (e.g., tuples) or lists of integers

```
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

```
A(ij) = 3.0;
A(i, j) = 4.0;
```

Array Arguments and Aliases

- Arrays are passed by reference by default

```
proc zero(X: []) { X = 0; }

zero(A[InnerD]);  // zeroes the inner values of A
```

- Formal array arguments can reindex actuals

```
proc f(X: [1..b,1..b]) { … } // X uses 1-based indices

f(A[lo..#b, lo..#b]);
```

- Array alias declarations provide similar functionality

```
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 in

```
sin(A)    ≈  forall a in A do sin(a)
2*A          forall a in A do 2*a
```

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

```
foo(Sparse)  ≈  forall i in Sparse do foo(i)
```

Multiple arguments promote using zipper promotion

```
pow(A, B)  ≈  forall (a,b) in (A,B) do pow(a,b)
```

Data Parallelism is Implicit

- 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

How Much Parallelism?

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

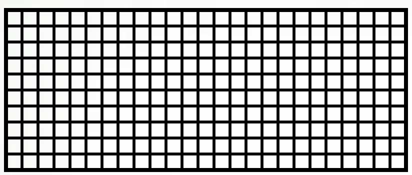
Outline

- Domains and Arrays
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Reductions and Scans
- Jacobi Iteration Example

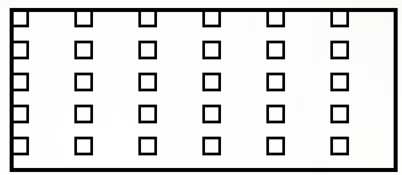
Chapel Domain Types

Chapel supports several domain types…

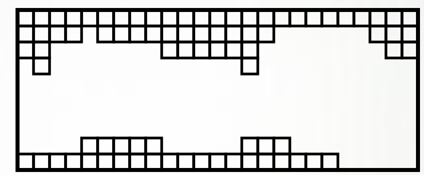
```
var OceanSpace = [0..#lat, 0..#long],
    AirSpace = OceanSpace by (2,4),
    IceSpace: sparse subdomain(OceanSpace) = genCaps();
```



dense

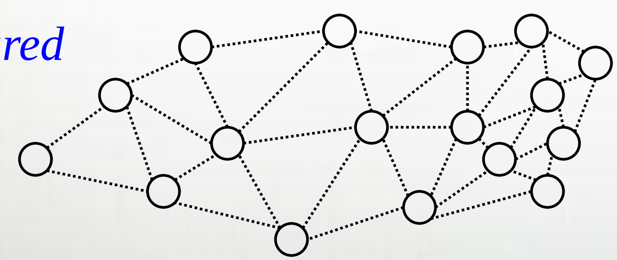


strided

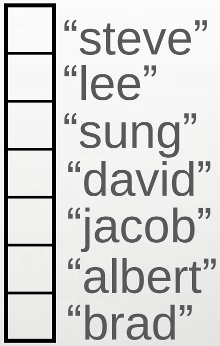


sparse

unstructured



associative

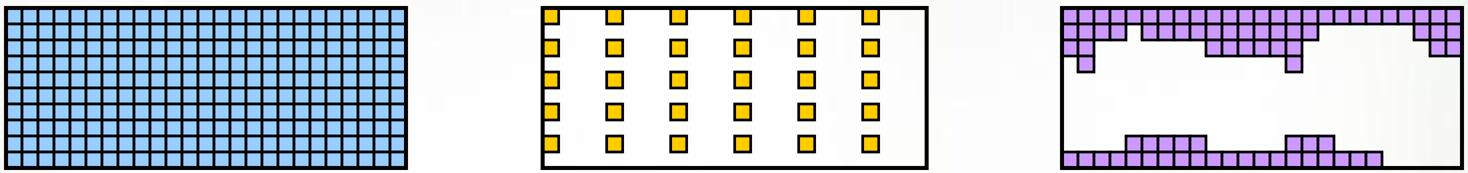


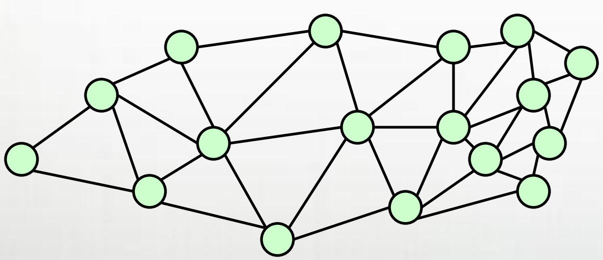

```
var Vertices: domain(opaque) = …,   People: domain(string) = …;
```

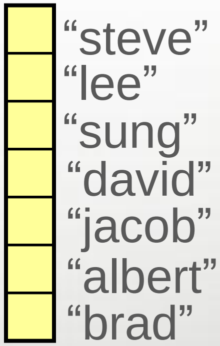
Chapel Array Types

All domain types can be used to declare arrays…

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





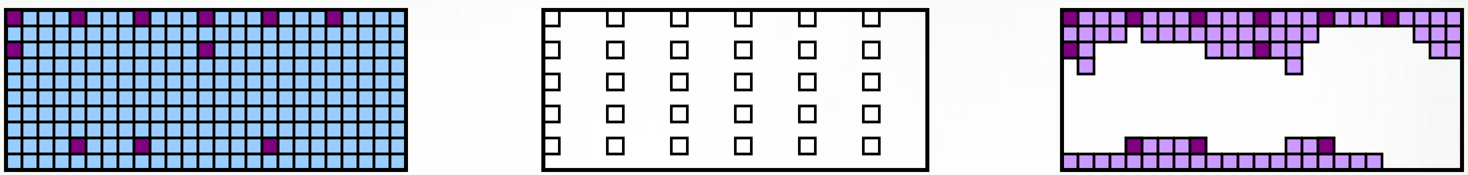


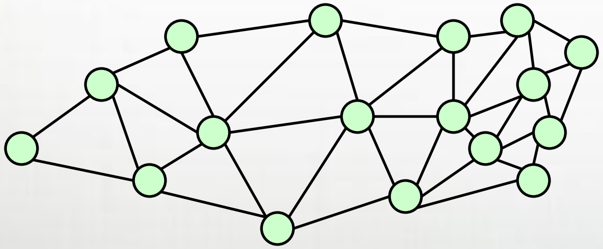

```
var Weight: [Vertices] real,           Age: [People] int;
```

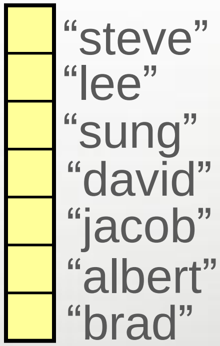
Iteration

…to iterate over index sets…

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







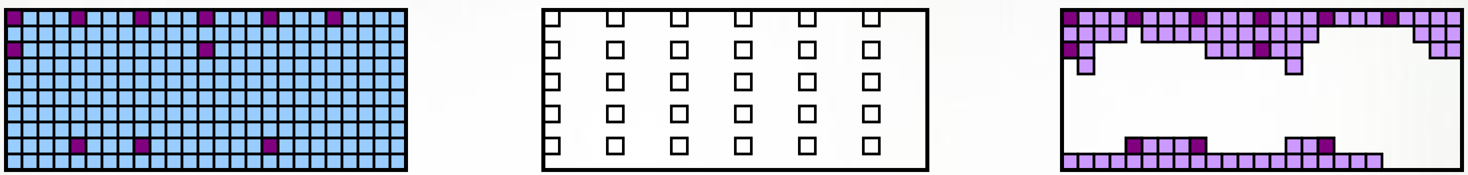

```
forall v in Vertices do
  Weight[v] = numEdges[v];
```

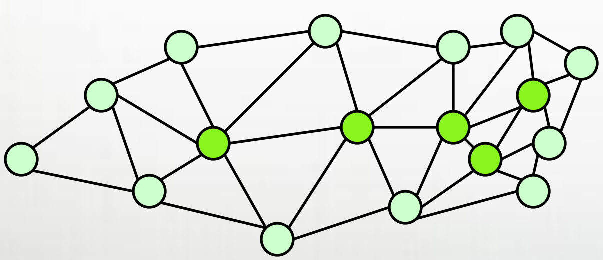
```
forall p in People do
  Age[p] += 1;
```

Slicing

…to slice arrays…

```
Ocean[AirSpace] += IceCaps[AirSpace];
```



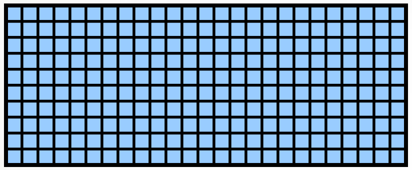


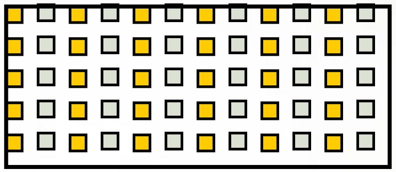
…`Vertices[Interior]`… …`People[Interns]`…

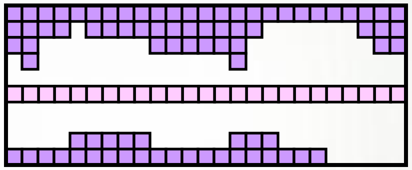
Reallocation

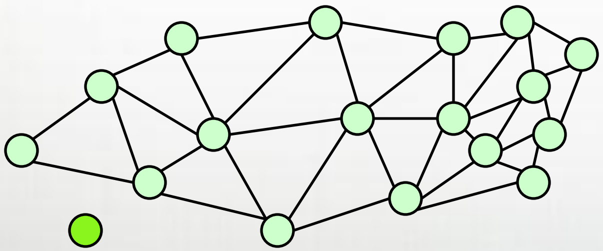
...and to reallocate arrays

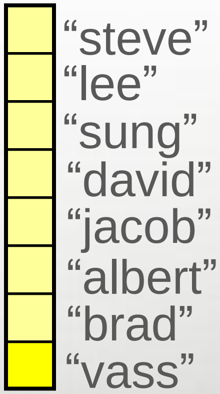
```
AirSpace = OceanSpace by (2,2);
IceSpace += genEquator();
```












```
newnode = Vertices.create();        People += “vass”;
```

Associative Domains and Arrays by Example

```
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;
```

Presidents
George
John
Thomas
James
Andrew
Martin
William

Birthday	Age
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
- 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

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

- Semantics
  - Computes parallel prefix over values using *scan-op*
  - *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

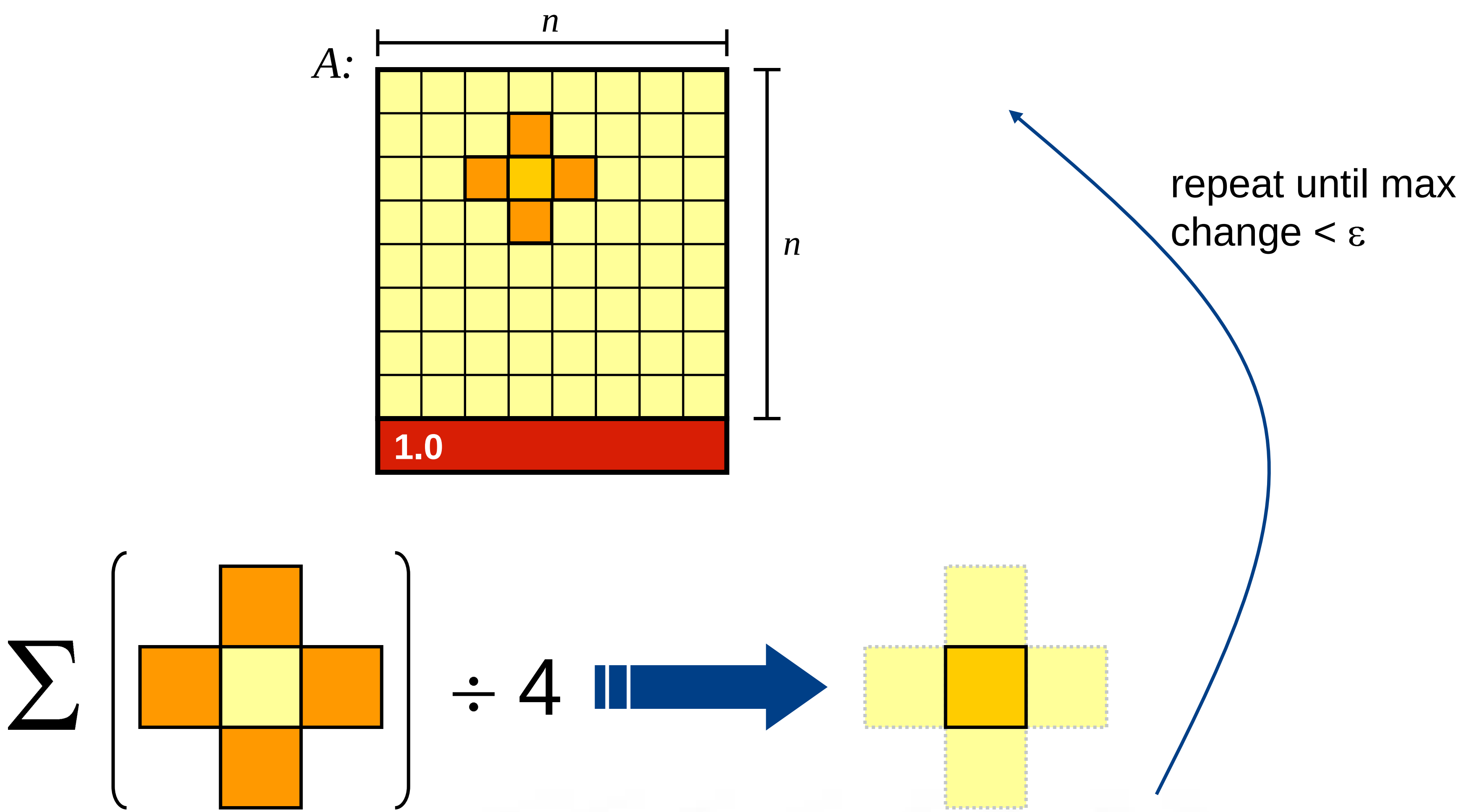
- Built-in
  - +, *, &&, ||, &, |, ^, min, max
  - minloc, maxloc
    - Takes a tuple of values and indices
    - Generates a tuple of the min/max value and its index
- User-defined
  - Defined via a class that supplies a set of methods
  - Compiler generates code that calls these methods
  - Based on:

    S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. *Global-view abstractions for user-defined reductions and scans*. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, 2006.

Outline

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

Jacobi Iteration in Pictures



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;

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

```
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[Las

do {
   [(i


   con
   A[D
} whi

writeln(A);
```

Declare program parameters

const ⇒ can't change values after initialization

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

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

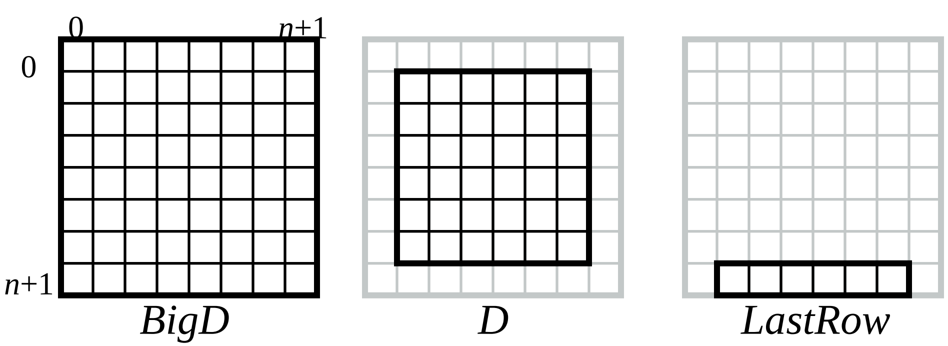
```
config const n = 6,
             epsilon = 1.0e-5;

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

<u>**Declare domains (first class index sets)**</u>

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

4;



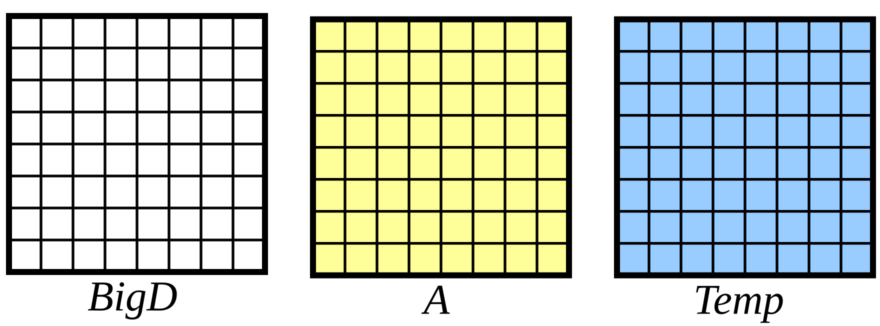
exterior ⇒ one of several built-in domain generators

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;
```

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)



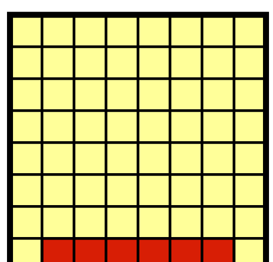

```
4;
```

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;
```

<u>Set Explicit Boundary Condition</u>

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation



A

Jacobi Iteration in Chapel

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

Compute 5-point stencil

[(*i,j*) in *D*] ⇒ parallel forall expression over *D*’s indices, binding them to new variables *i* and *j*


```
  [(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

```
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

```
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[La

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);
```

<u>Copy data back & Repeat until done</u>

uses slicing and whole array assignment
standard *do…while* loop construct

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;

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

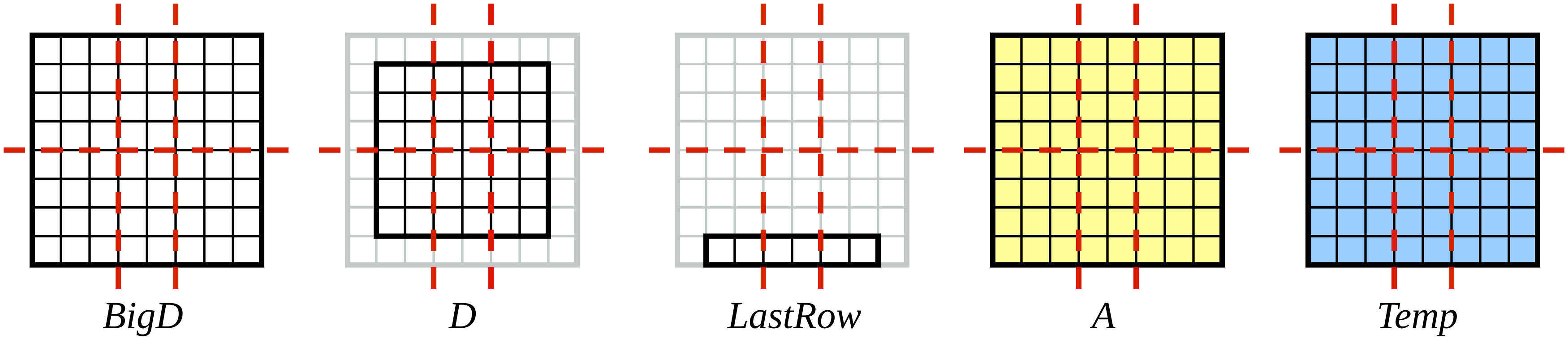
```
config const n = 6,
             epsilon = 1.0e-5;

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

```
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

Questions?

- 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