Chapel: Data Parallelism

Data vs. Task Parallelism (Our Terminology)

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 using distinct tasks
- 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

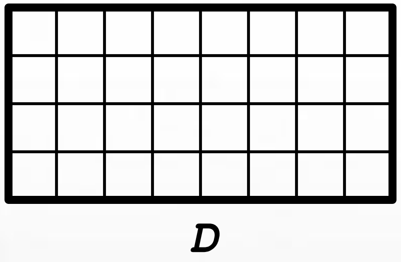
Domains

Domain: A first-class index set

- Fundamental Chapel concept for data parallelism
- A generalization of ZPL's *region* concept
- Domains may optionally be distributed

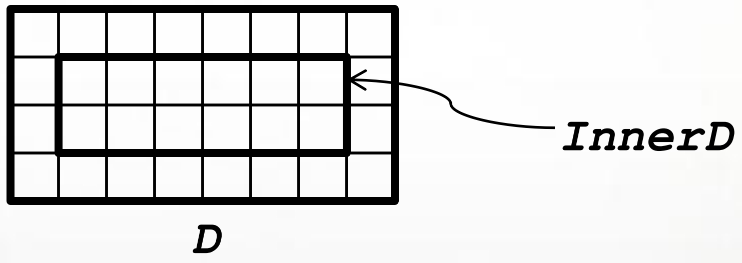
Sample Domains

```
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];
```



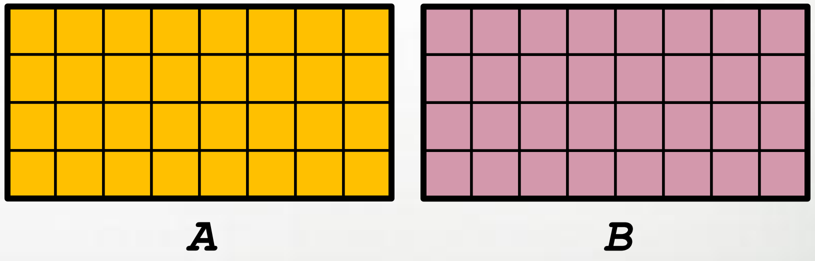
Domains Define Arrays

- Syntax

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

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

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



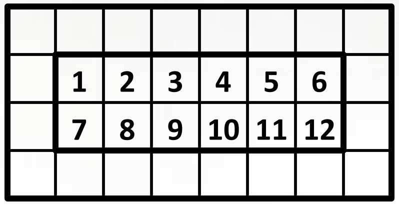
- Revisited example

```
var A: [1..3] int; // creates anonymous domain [1..3]
```

Domain Iteration

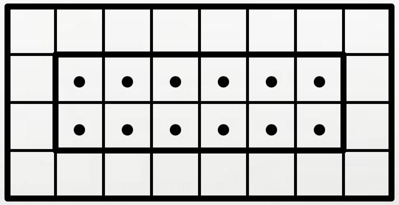
- For loops (discussed already)
  - Execute loop body once per domain index, serially

```
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...

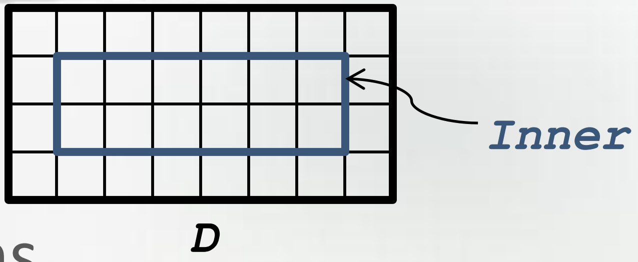
- Methods for creating new domains

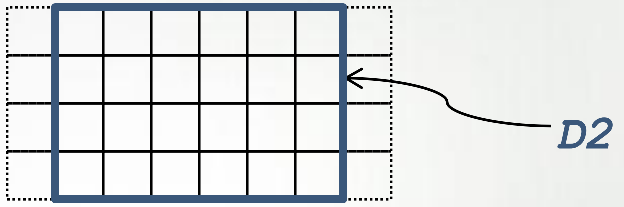
```
var D2 = InnerD.expand(1,0);
```

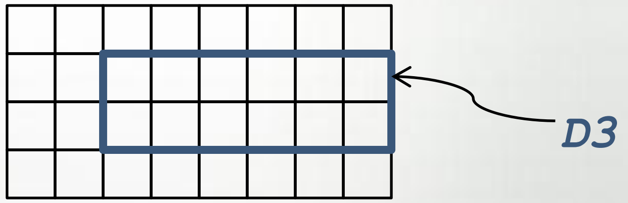
```
var D3 = InnerD.translate(0,1);
```

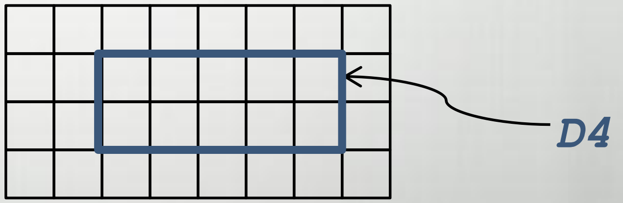
- Intersection via Slicing

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





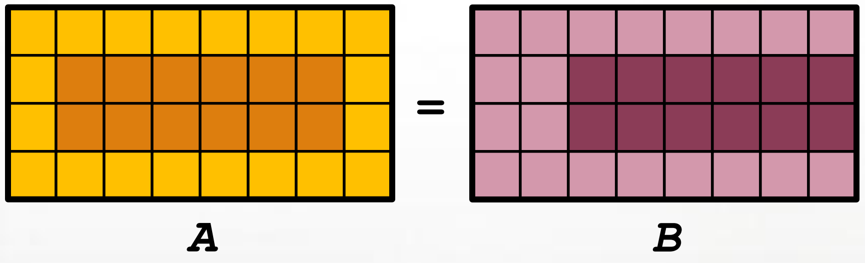




Array Slicing/Sub-Arrays

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

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

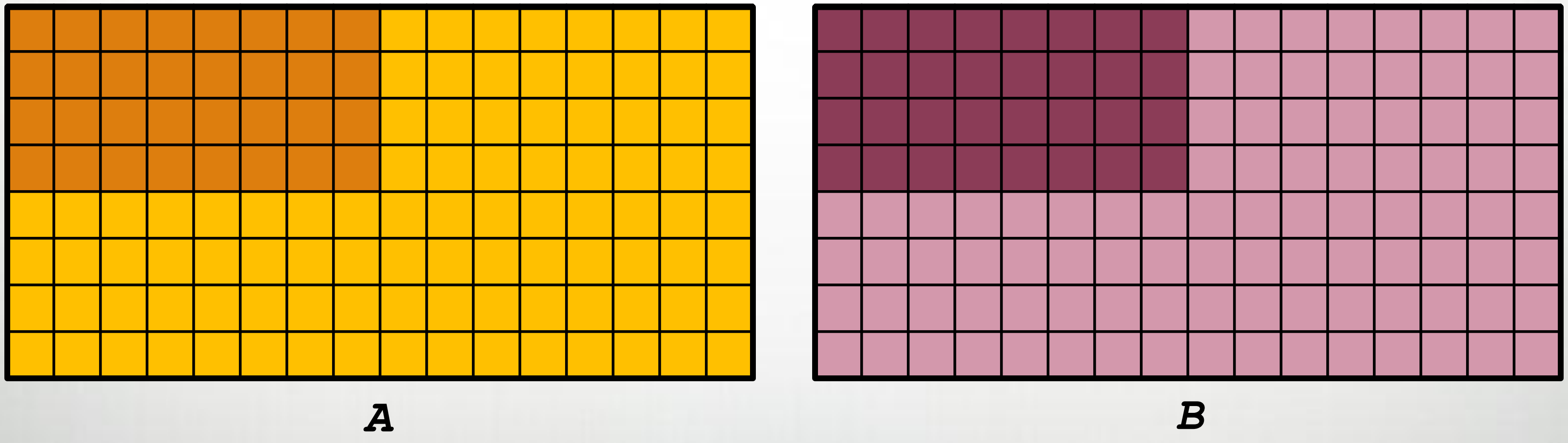


Array Reallocation

Reassigning a domain logically reallocates its arrays

- values are preserved for common indices

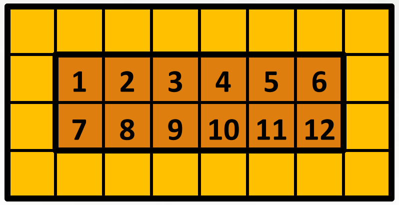
```
D = [1..2*m, 1..2*n];
```



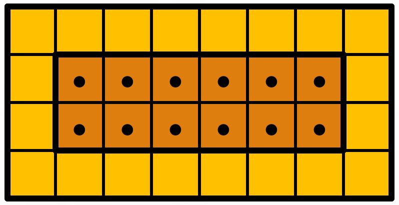
Array Iteration

- Array expressions also support for and forall loops

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

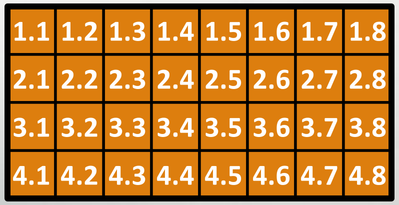


```
forall a in A[InnerD] do ...
```



- Array loop variables refer to array values (modifiable)

```
forall (a, (i,j)) in (A, D) do a = 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

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;
```

Promoted Functions and Operators

Functions/operators expecting scalars can also take...

- ...arrays, causing each element to be passed in

```
sin(A)
2*A
```

≈

```
forall a in A do sin(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)
```

Where is the Parallelism?

- 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

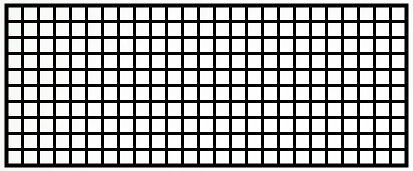
Outline

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

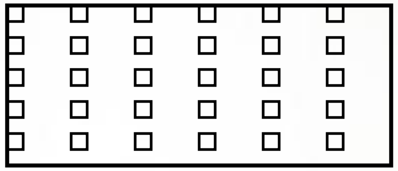
Chapel Domain Types

Chapel supports several domain types…

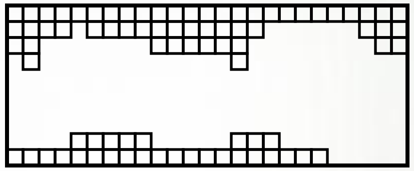
```
var OceanSpace = [0..#lat, 0..#long],
    AirSpace = OceanSpace by (2,4),
    IceSpace: sparse subdomain(OceanSpace) = genCaps();
```



dense

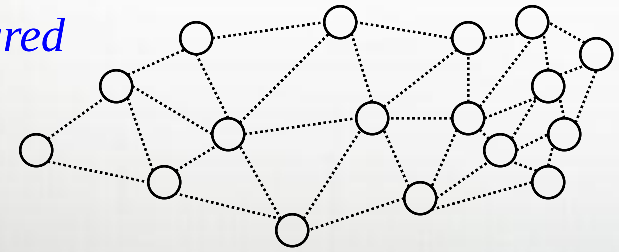


strided

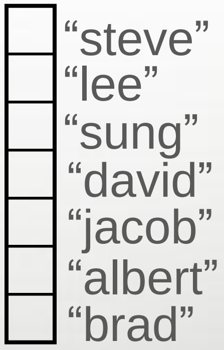


sparse

unstructured



associative

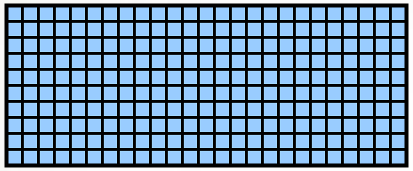
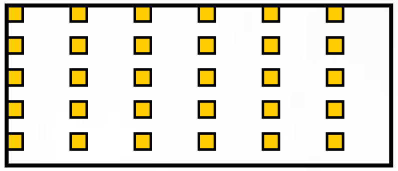
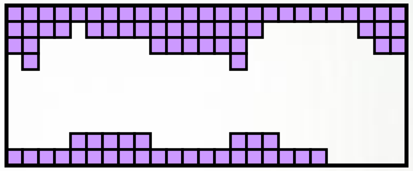


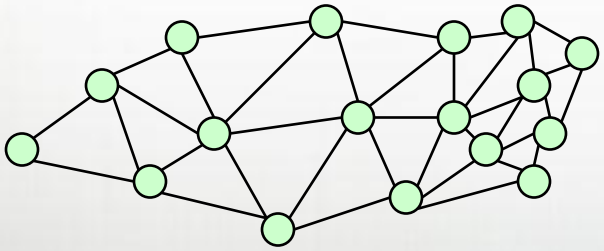
```
var Vertices: domain(opaque) = …,   People: domain(string) = …;
```

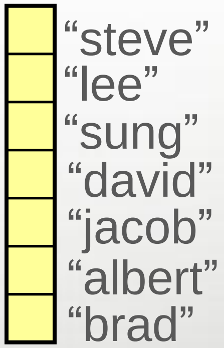
Chapel Array Types

All domain types can be used to declare arrays…

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



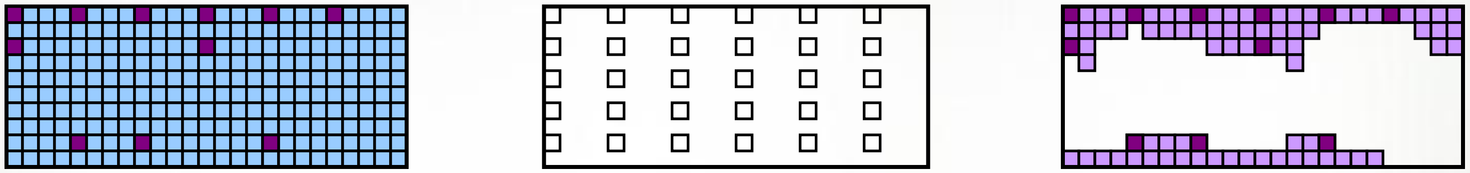


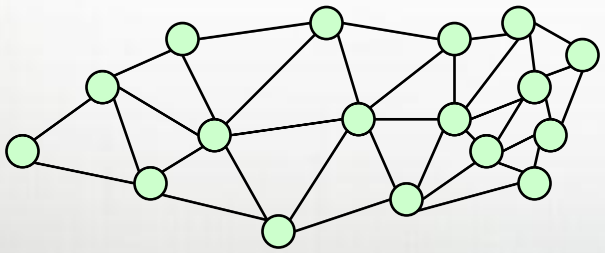
```
var Weight: [Vertices] real,        Age: [People] int;
```

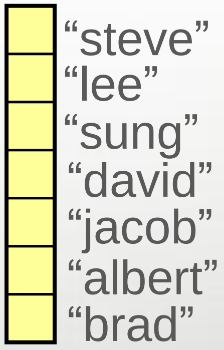
Iteration

...to iterate over index sets...

```
forall ij in AirSpace do
  Ocean(ij) += IceCaps(ij);
```







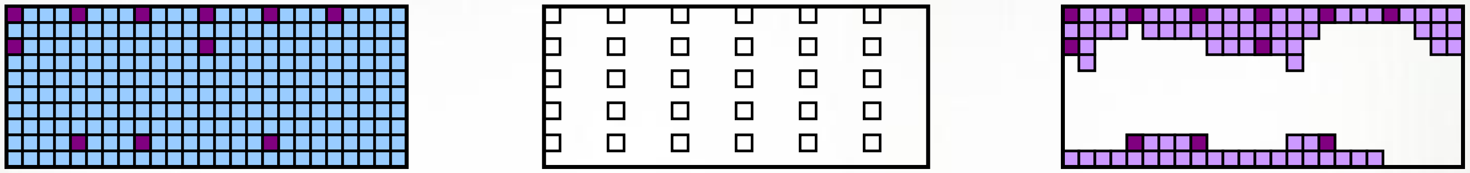
```
forall v in Vertices do
  Weight(v) = numEdges(v);
```

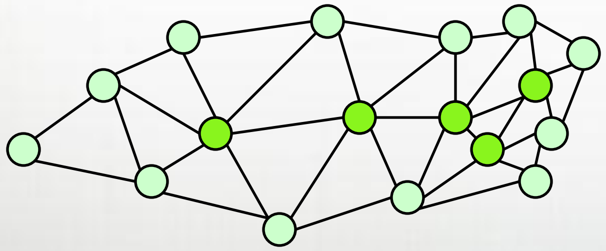
```
forall p in People do
  Age(p) += 1;
```

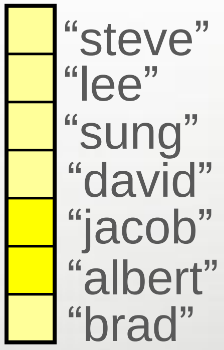
Slicing

…to slice arrays…

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





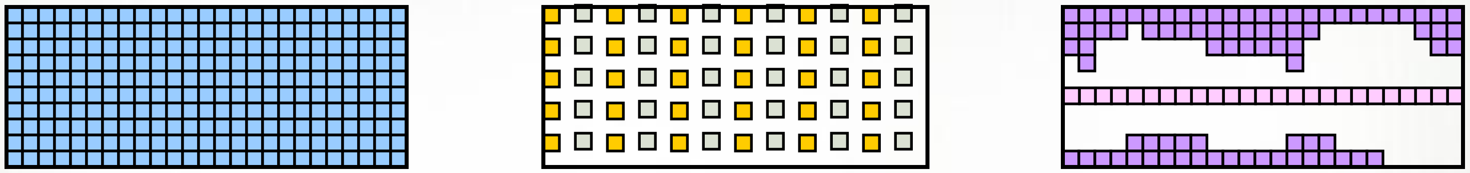


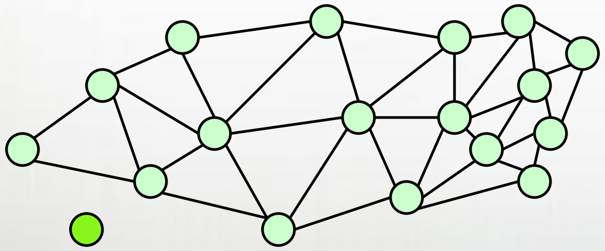
…Vertices[Interior]… …People[Interns]…

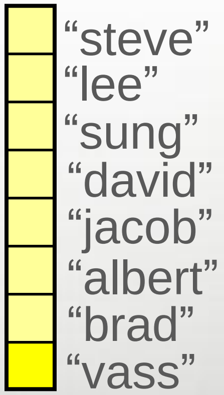
Reallocation

…and to reallocate arrays

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



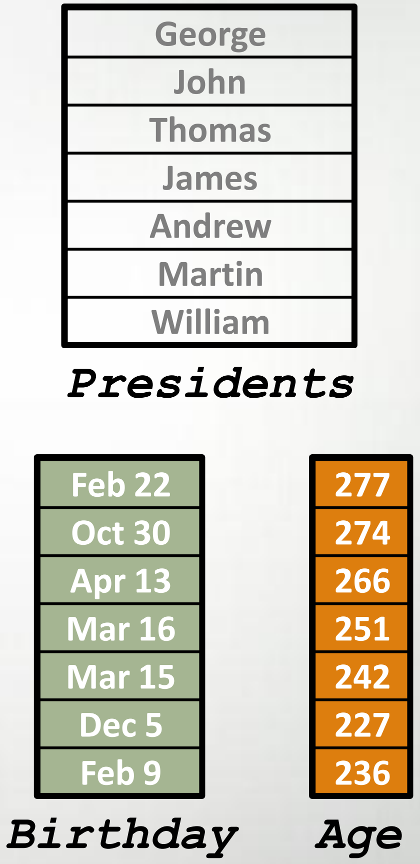




```
newnode = Vertices.create();    People += "vass";
```

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;
```



Outline

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

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);
```

Reduction and Scan Operators

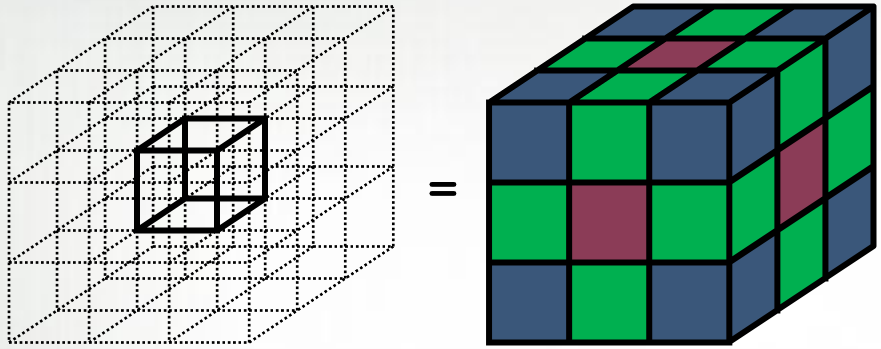
- Built-in
  - +, *, &&, ||, &, |, ^, min, max
  - minloc, maxloc
    - Takes a tuple of values and indices
    - Generates a tuple of the min/max value and its index
- User-defined
  - Defined via a class that supplies a set of methods
  - Compiler generates code that calls these methods
  - Based on:

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
- NAS MG Stencil Example

Revisiting the *rprj3* Stencil from NAS MG

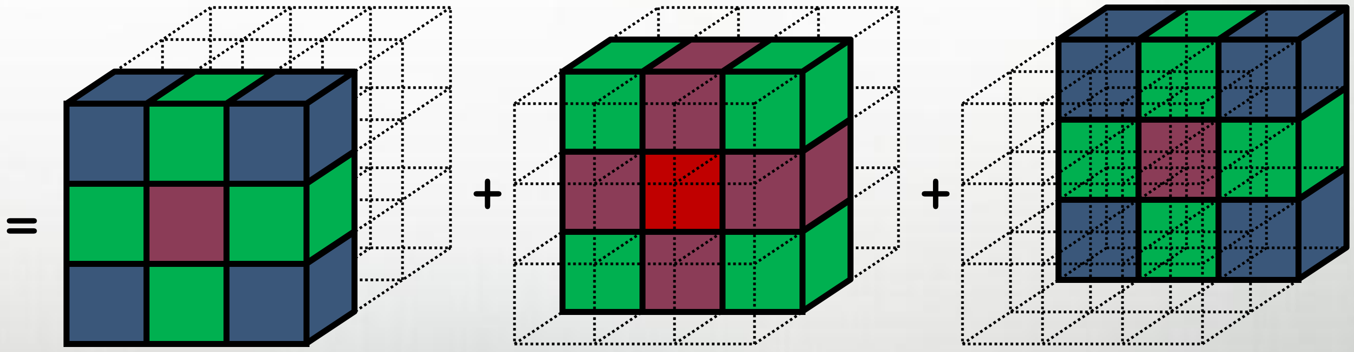


🟥 = w_0

🟪 = w_1

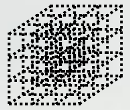
🟩 = w_2

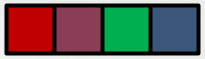
🟦 = w_3

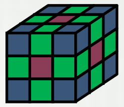


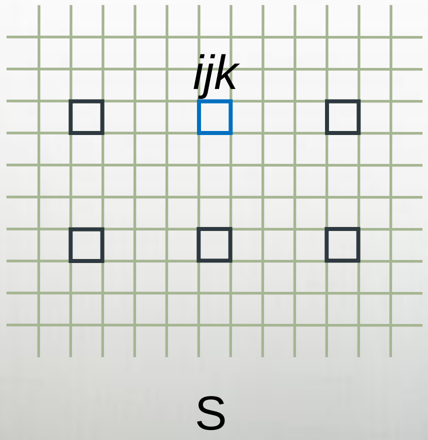
NAS MG Stencil in Chapel Revisited

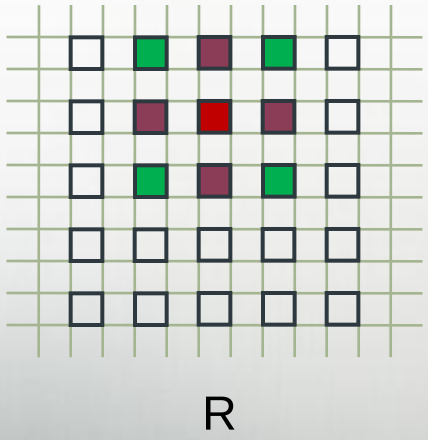
```
proc rprj3(S: [?SD], R: [?RD]) {
  const Stencil = [-1..1, -1..1, -1..1],
        W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
        W3D = [(i,j,k) in Stencil] W[(i!=0) + (j!=0) + (k!=0)];

  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 and irregular domains/arrays generating parallelism
- Scalar performance lacking in some cases (particularly higher-dimensional operations)
- Implementation of unstructured domains/arrays is correct but inefficient

Future Directions

- Gain more experience with unstructured (graph-based) domains and arrays

Questions?