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

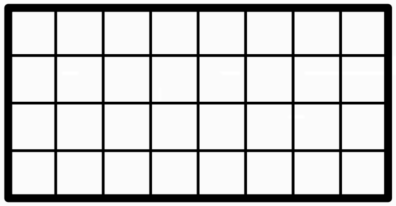
Domain: A first-class index set

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

Sample Domains

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

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



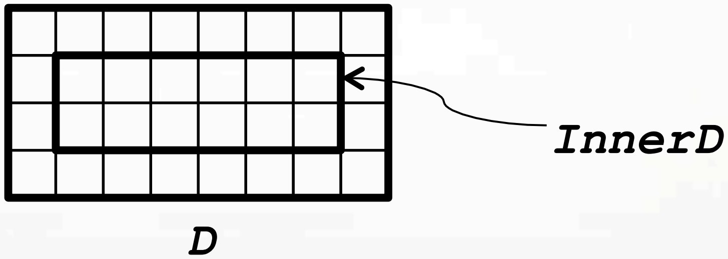
D

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];
```



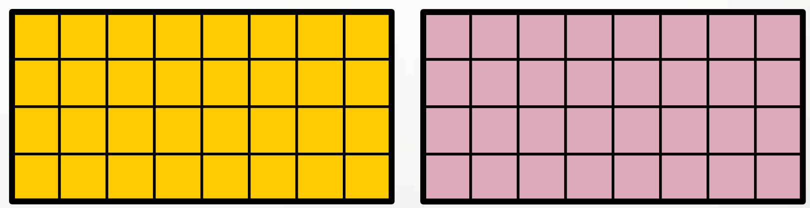
Domains Define Arrays

- Syntax

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

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

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



A ***B***

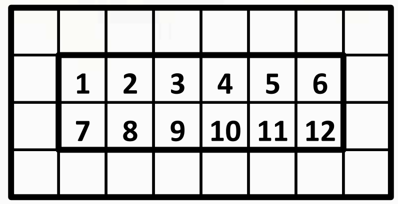
- Revisited example

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

Domain Iteration

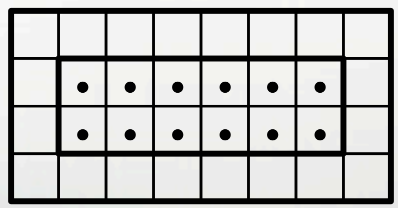
- For loops (discussed already)
  - Execute loop body once per domain index, serially
  - Index variable takes on const index values

```
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 domain index values (`const`)

Other Forall Loops

Forall loops also support...

- A shorthand:

```
[(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

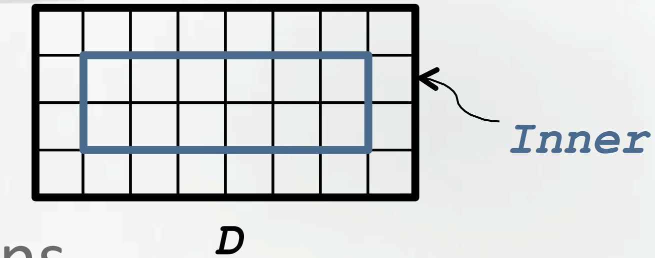
```
var D2 = InnerD.expand(1,0);
```

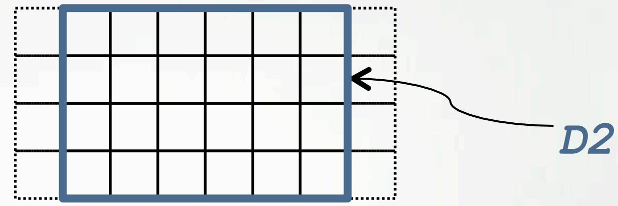
- Overloaded Operators

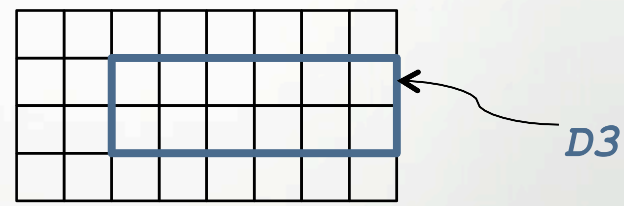
```
var D3 = InnerD + (0,1);
```

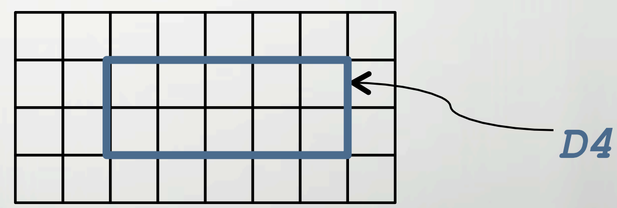
- Intersection via Slicing

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





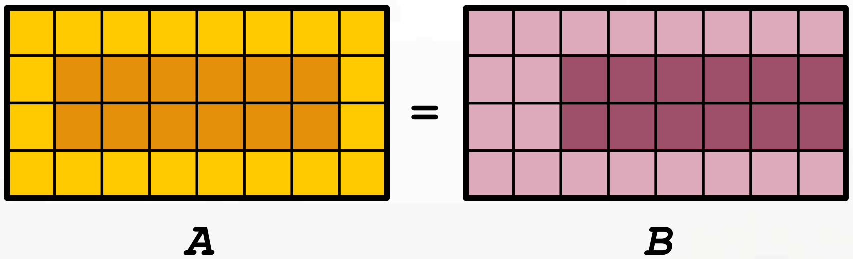




Array Slicing/Sub-Arrays

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

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

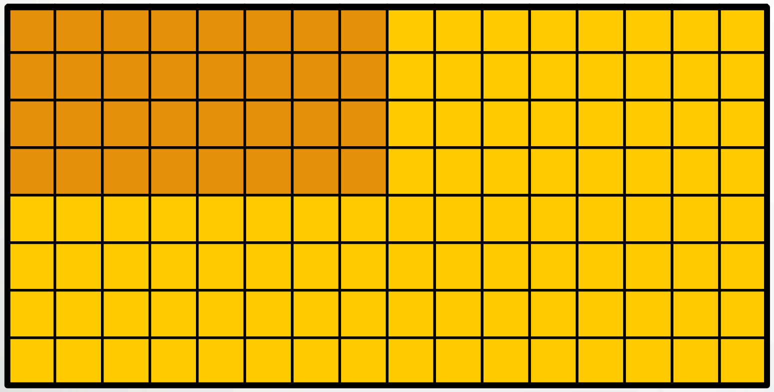


Array Reallocation

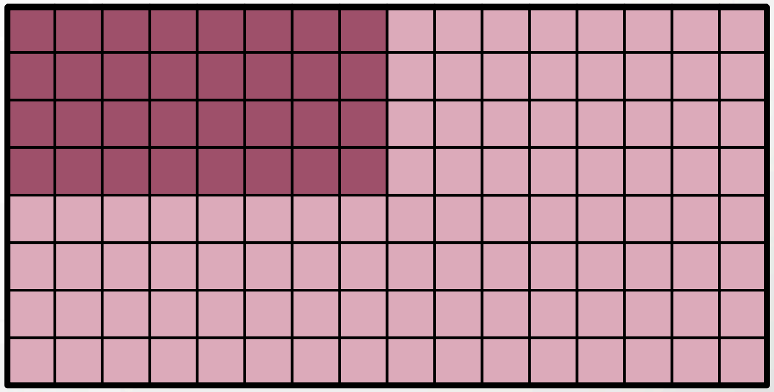
Reassigning a domain logically reallocates its arrays

- values are preserved for common indices

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



A

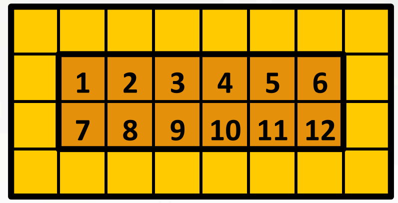


B

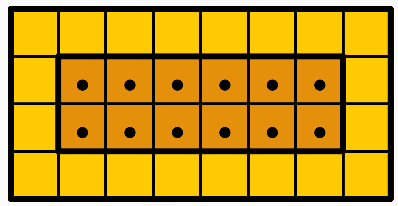
Array Iteration

- Array expressions also support for and forall loops

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

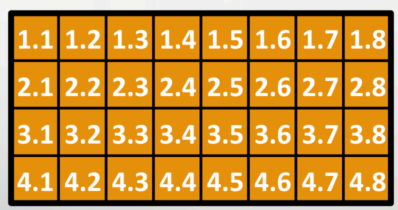



```
forall a in A[InnerD] do ...
```



- Array loop variables refer to array values (modifiable)

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



Array Arguments and Aliases

- Array values are passed by reference

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

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

- Formal array arguments can reindex actuals

```
def 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

```
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

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

Multiple arguments can promote using either...

- Zipper promotion

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

- Tensor product promotion

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

How Much Parallelism?

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

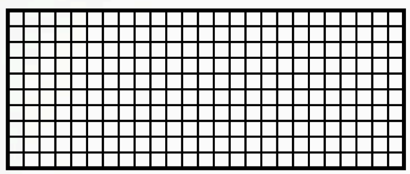
Outline

- Domains and Arrays
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Reductions and Scans
- NAS MG Stencil Revisited

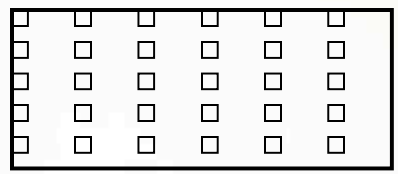
Chapel Domain Types

Chapel supports several domain types…

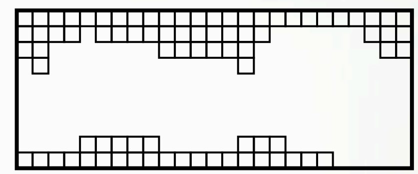
```
var OceanSpace = [0..#lat, 0..#long],
    AirSpace = OceanSpace by (2,4),
    IceSpace: sparse subdomain(OceanSpace) = genCaps();
```



dense

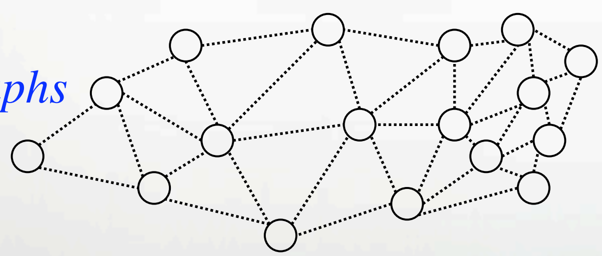


strided



sparse

graphs



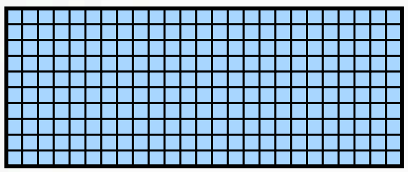
associative

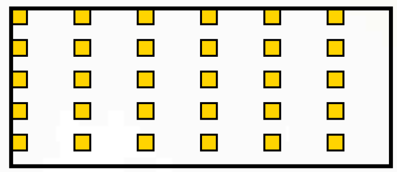

```
var Vertices: domain(opaque) = …,   People: domain(string) = …;
```

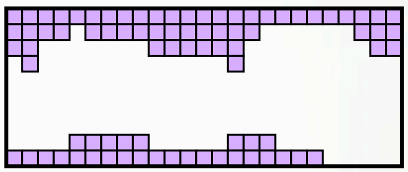
Chapel Array Types

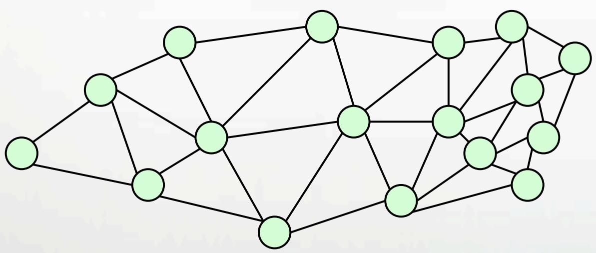
All domain types can be used to declare arrays…

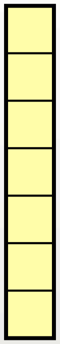
```
var Ocean: [OceanSpace] real,
    Air: [AirSpace] real,
    IceCaps[IceSpace] real;
```









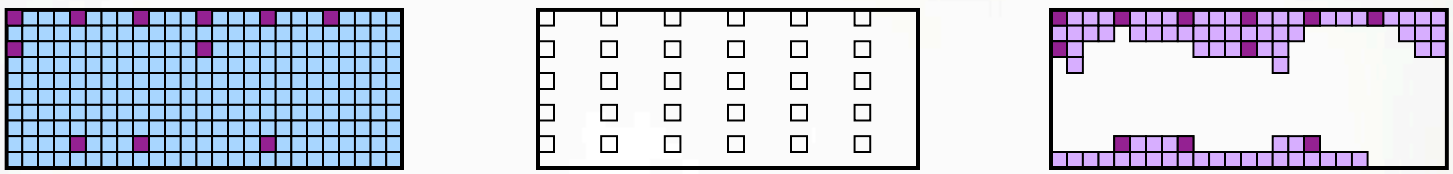


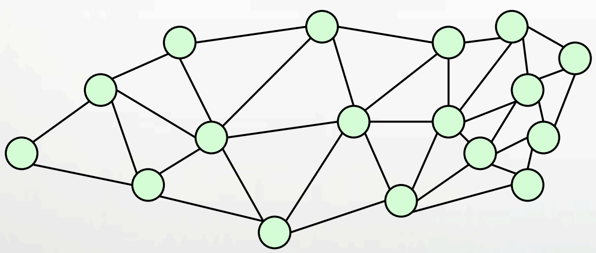

```
var Weight: [Vertices] real,          Age: [People] int;
```

Iteration

…to iterate over index sets…

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





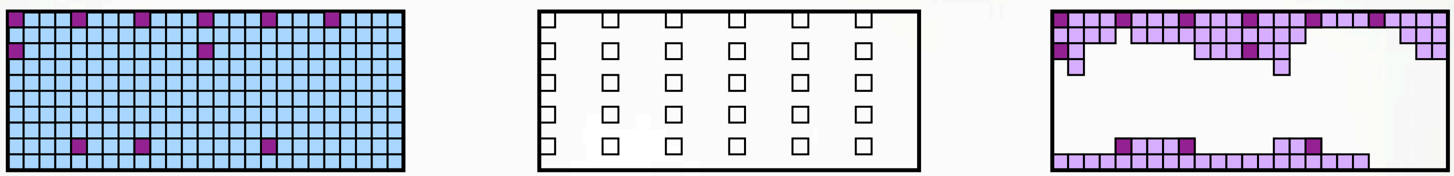

```
forall v in Vertices do
  Weight(v) = numEdges(v);
```

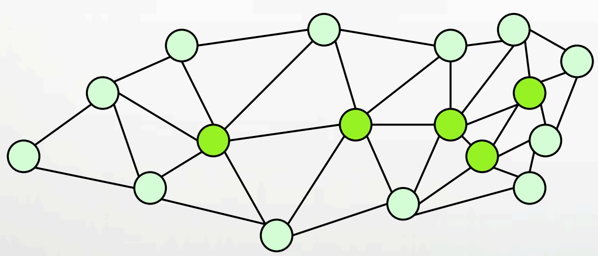
```
forall p in People do
  Age(p) += 1;
```

Slicing

…to slice arrays…

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





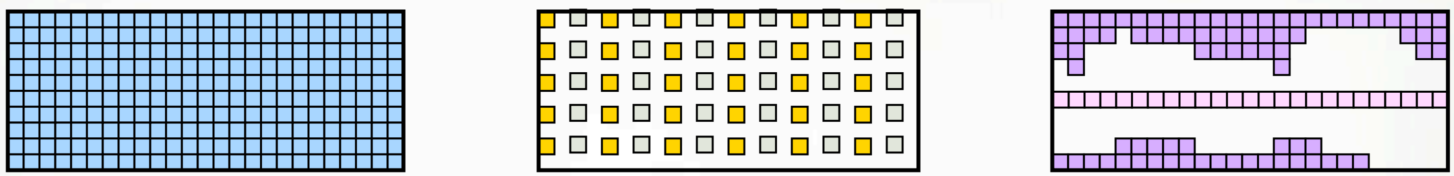
…`Vertices[Interior]`…

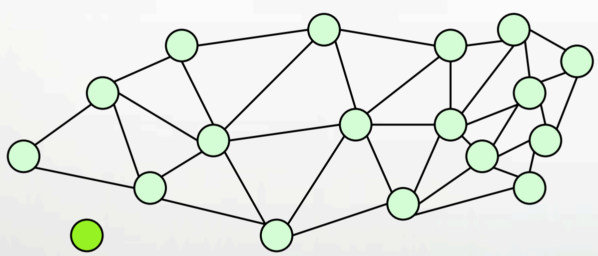
…`People[Interns]`…

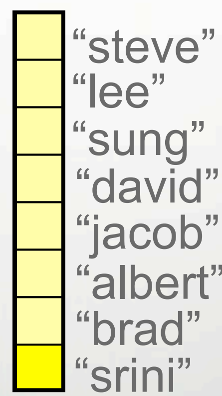
Reallocation

…and to reallocate arrays

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








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

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
  - Reductions
  - Scans
- 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);
```

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

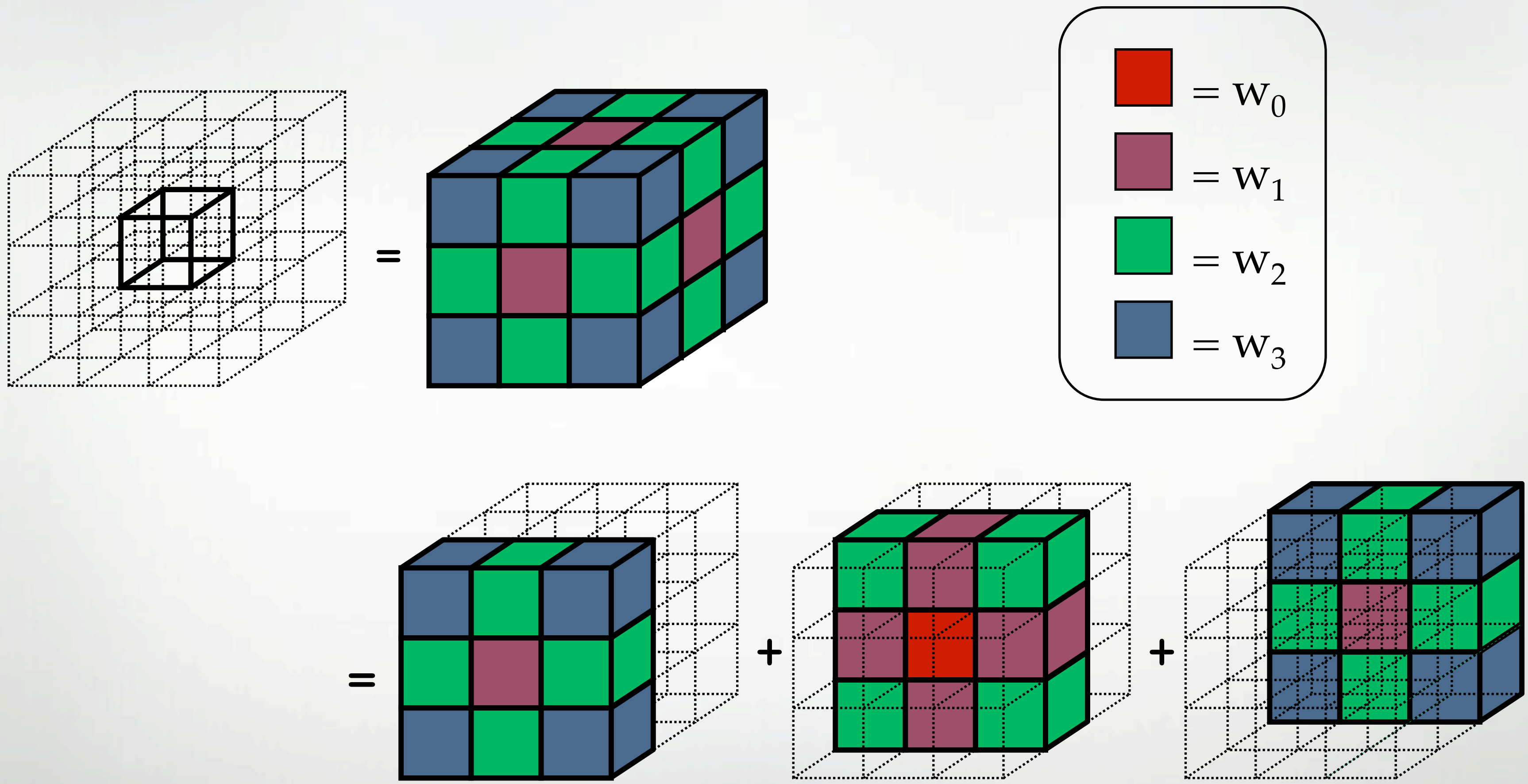
- Built-in
  - +, *, &&, ||, &, |, ^, min, max
  - minloc, maxloc
    - Takes a tuple of values and indices
    - Generates a tuple of the min/max value and its index
- User-defined
  - Defined via a class that supplies a set of methods
  - Compiler generates code that calls these methods
  - Based on:

    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
- NAS MG Stencil Revisited

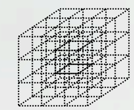
Revisiting the *rprj3* Stencil from NAS MG

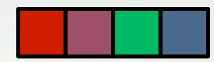


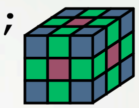
NAS MG Stencil in Chapel Revisited

```
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