Chapel: Data Parallelism

Sung-Eun Choi and Steve Deitz

Cray Inc.

Outline

- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
- Data Parallel Operations
- Examples

Domains

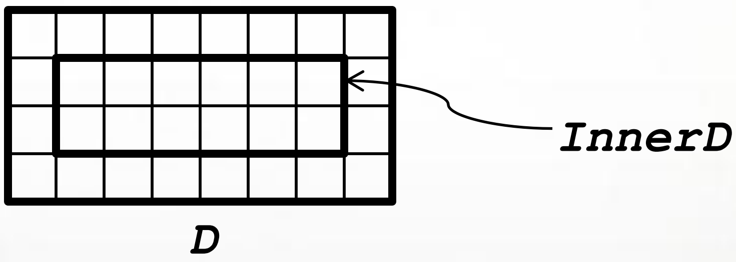
- A first-class index set
  - Specifies size and shape of arrays
  - Supports iteration, array operations
  - Potentially distributed across locales
- Three main classes
  - Arithmetic—indices are Cartesian tuples
  - Associative—indices are hash keys
  - Opaque—indices are anonymous
- Fundamental Chapel concept for data parallelism
- A generalization of ZPL's region concept

Sample Arithmetic Domains

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

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

var InnerD: domain(2) = [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;
```

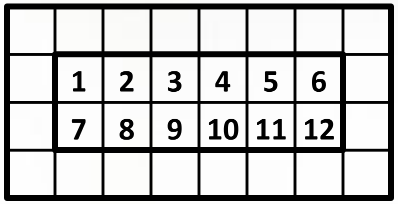
- Revisited example

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

Domain Iteration

- For loops (discussed already)
  - Executes loop body once per loop iteration
  - Order is serial

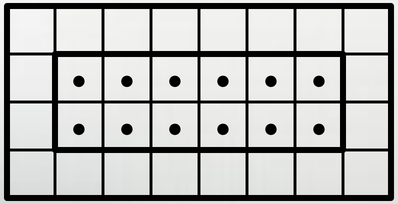
```
for i in InnerD do ...
```



D

- Forall loops
  - Executes loop body once per loop iteration
  - Order is parallel (must be *serializable*)

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



D

Other Forall Loops

Forall loops also support...

- A shorthand:

```
[(i,j) in D] A(i,j) = 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

- An expression-based form:

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

- A shorthand expression-based form:

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

Usage of For, Forall, and Coforall

- Use for when
  - A loop must be executed serially
  - One task is sufficient for performance
- Use forall when
  - The loop can be executed in parallel
  - The loop can be executed serially
  - Degree of concurrency << # of iterations
- Use coforall when
  - The loop must be executed in parallel
    (And not just for performance reasons!)
  - Each iteration has substantial work

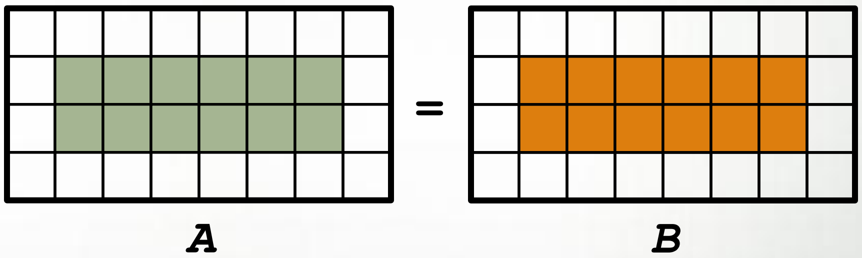
Data Parallelism Configuration Constants

- **--dataParTasksPerLocale=#**
  - Specify # of tasks to execute forall loops
  - Default: number of cores *(in current implementation)*
- **--dataParIgnoreRunningTasks=[true|false]**
  - If false, reduce # of forall tasks by # of running tasks
  - Default: true *(in current implementation)*
- **--dataParMinGranularity=#**
  - If > 0, reduce # of forall tasks if any task has fewer iterations
  - Default: 0 *(in current implementation)*

Other Domain Functionality

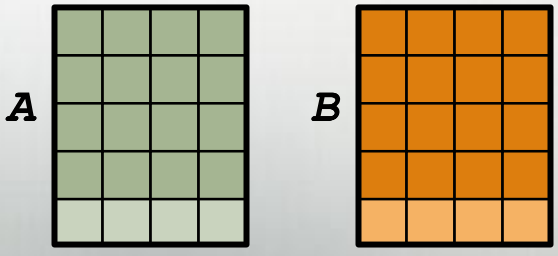
- Domain methods (exterior, interior, translate, ...)
- Domain slicing (intersection)
- Array slicing (sub-array references)

```
A(InnerD) = B(InnerD);
```



- Array reallocation
  - Reassign domain → change array
  - Values are preserved (new elements initialized)

```
D = [1..m+1, 1..m];
```



Array Arguments and Aliases

- Arrays are passed by reference

```
def f(A: []) { A = 0; }

f(A[InnerD]);
```

- Non-argument array alias of a slice

```
var AA => A(InnerD);
```

- Re-indexing arrays

```
def f(A: [1..n-2,1..m-2]);

f(A[2..n-1,2..m-1]);
```

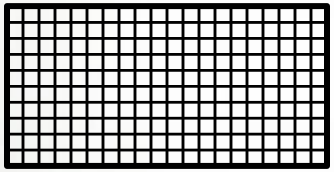
```
var AA: [1..n-2,1..m-2] => A[2..n-1,2..m-1];
```

Outline

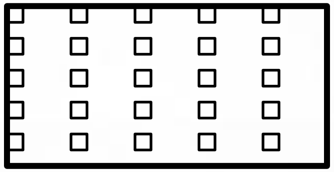
- Domains and Arrays
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
- Examples

The Varied Kinds of Domains

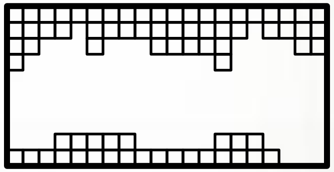
```
var Dense: domain(2) = [1..10, 1..20],
    Strided: domain(2) = Dense by (2, 4),
    Sparse: sparse subdomain(Dense) = genIndices(),
    Associative: domain(string) = readNames(),
    Opaque: domain(opaque);
```



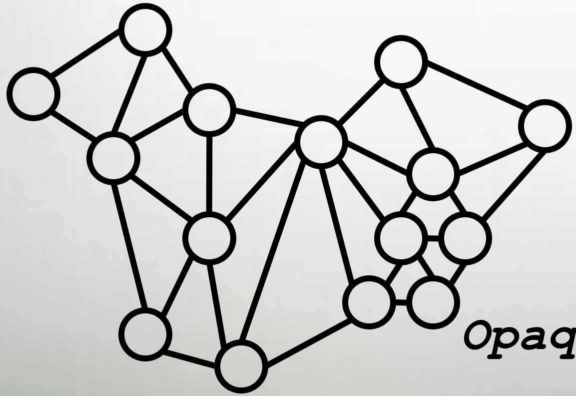
Dense



Strided



Sparse

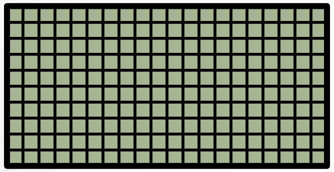


Opaque

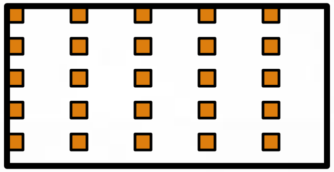
Associative

The Varied Kinds of Arrays

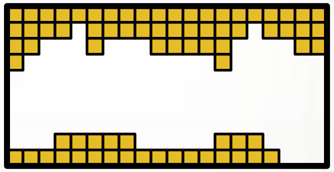
```
var DenseArr: [Dense] real,
    StridedArr: [Strided] real,
    SparseArr: [Sparse] real,
    AssociativeArr: [Associative] real,
    OpaqueArr: [Opaque] real;
```



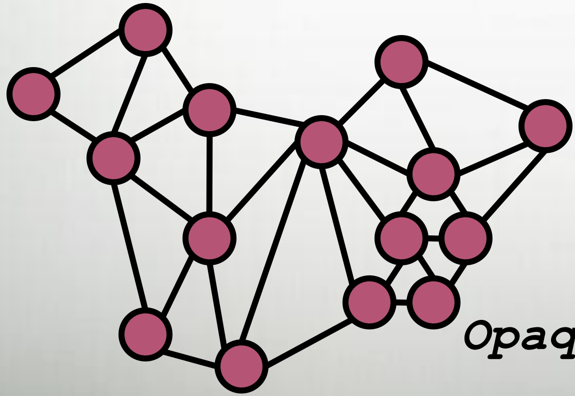
DenseArr



StridedArr



SparseArr

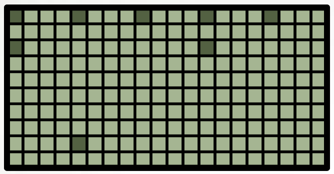


OpaqueArr

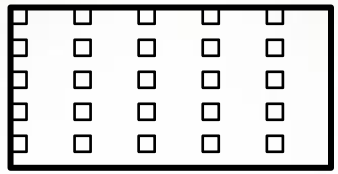
AssociativeArr

All Domains Support Iteration

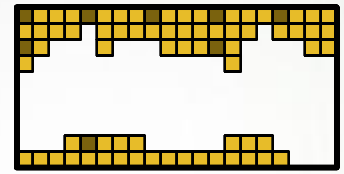
```
forall (i,j) in Strided {
  DenseArr(i,j) += SparseArr(i,j);
}
```



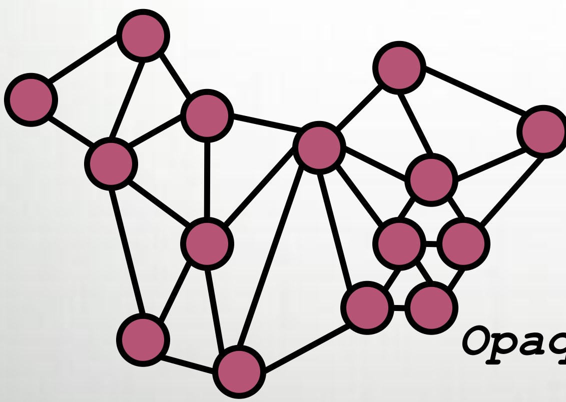
DenseArr



Strided



SparseArr



OpaqueArr

AssociativeArr

(Also, all domains support slicing, reallocation, ...)

Associative Domains and Arrays by Example

```
var Presidents: domain(string) =
      (“George”, “John”, “Thomas”,
       “James”, “Andrew”, “Martin”);

Presidents += “William”;


var Ages: [Presidents] int,
    Birthdays: [Presidents] string;

Birthdays(“George”) = “Feb 22”;


forall president in Presidents do
  if Birthdays(president) == today then
    Ages(president) += 1;
```

Presidents
George
John
Thomas
James
Andrew
Martin
William

Birthdays	Ages
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
- Data Parallel Operations
  - Promotion
  - Reductions
  - Scans
- Examples

Data Parallel Promotion

Functions/operators expecting scalars can also take...

- Arrays, causing each element to be passed

```
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

```
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)
```

Reductions

- Syntax

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

- Semantics
  - Combines iterated elements with *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));
```

Scans

- Syntax

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

- Semantics
  - Computes parallel prefix of *scan-op* over elements
  - *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

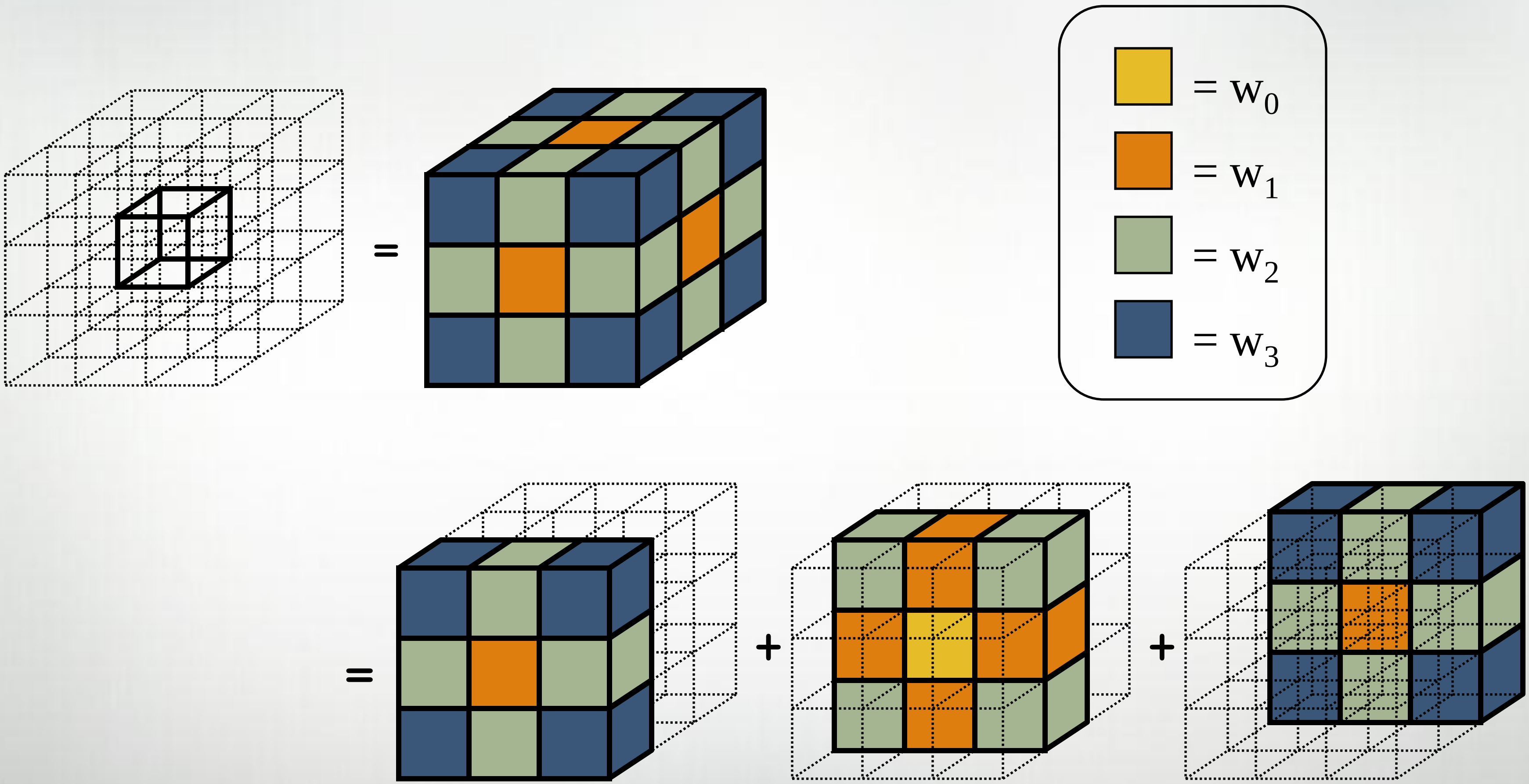
    (Generate a tuple of the min/max and its index)
- User-defined
  - Defined via a class that supplies a set of methods
  - Compiler generates code that calls these methods
  - More information:

    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
- Data Parallel Operations
- Examples
  - NAS MG stencil revisited
  - Pre-key copy kernel
  - A = B + C

NAS MG Stencil Revisited

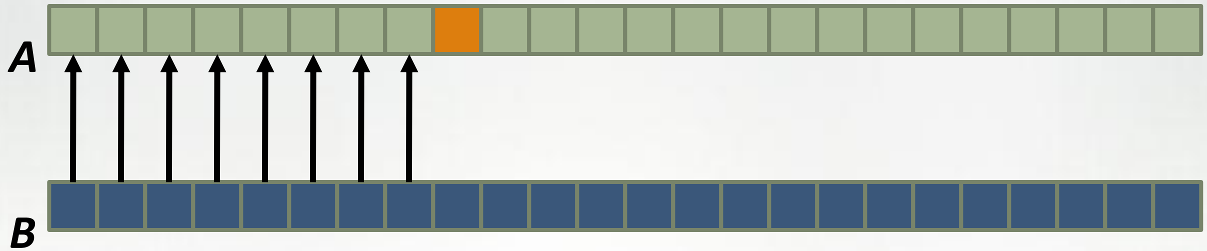


NAS MG Stencil in Chapel Revisited

```
def rprj3(S, R) {
  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 inds in S.domain do
    S(inds) =
      + reduce [offset in Stencil] (W3D(offset) *
                                     R(inds + offset*R.stride));
}
```

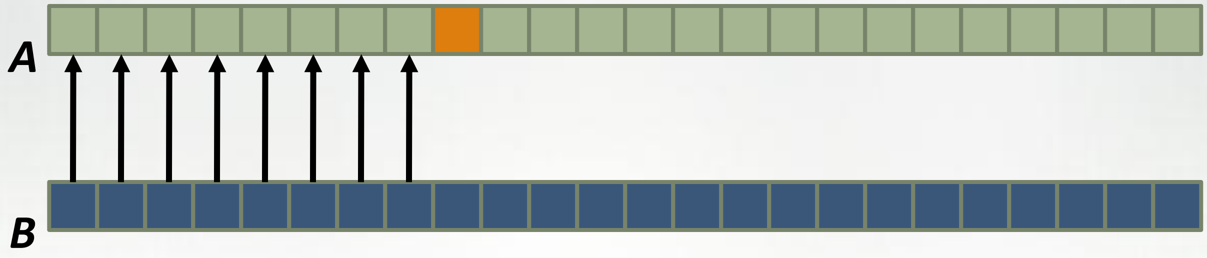
Pre-Key Copy




```
var A, B: [1..n] real, key: real;

for i in 1..n {
  if A(i) == key then
    break;
  A(i) = B(i);
}
```

First Parallelization Attempt

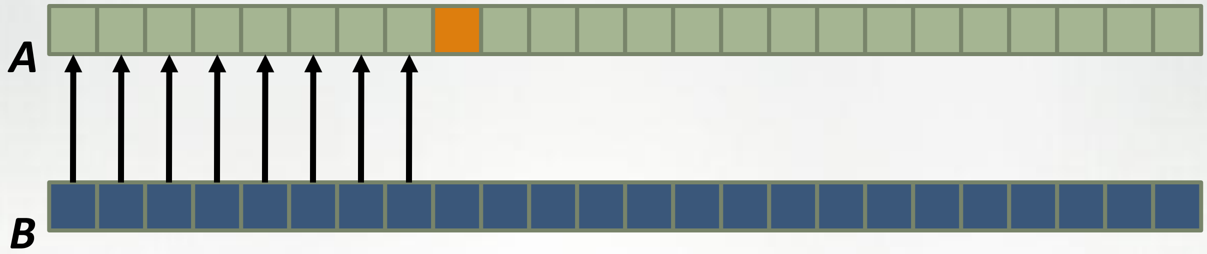



```
var A, B: [1..n] real, key: real;

forall i in 1..n {
  if A(i) == key then
    break;
  A(i) = B(i);
}
```

```
error: break is not allowed in forall statement
```

Pre-Compute Region to Copy

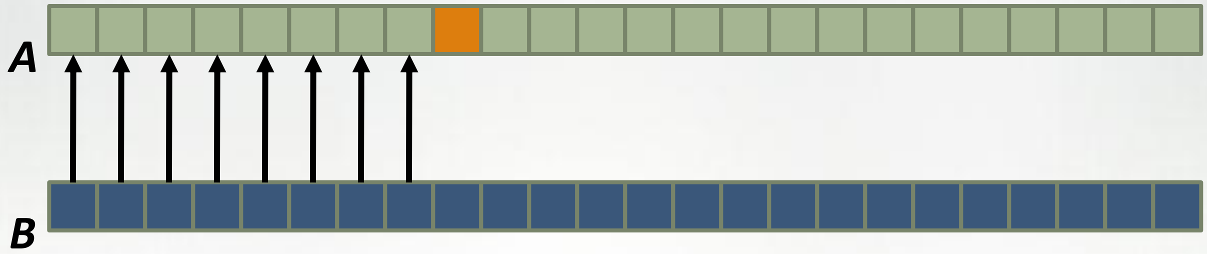



```
var A, B: [1..n] real, key: real;

var loc: int = n;
for i in 1..n do
  if A(i) == key {
    loc = i-1;
    break;
  }

forall i in 1..loc do
  A(i) = B(i);
```

Parallelize Pre-Computation

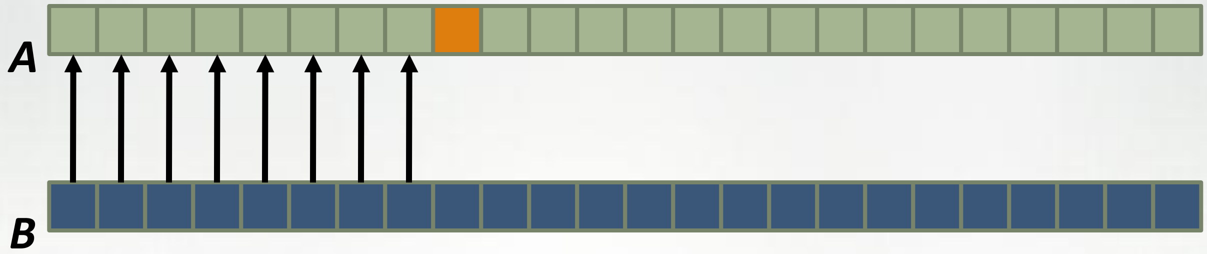



```
var A, B: [1..n] real, key: real;

var loc: sync int = n;
forall i in 1..n do
  if A(i) == key then
    loc = min(loc, i-1);

forall i in 1..loc do
  A(i) = B(i);
```

Parallelize Using maxloc Reduce

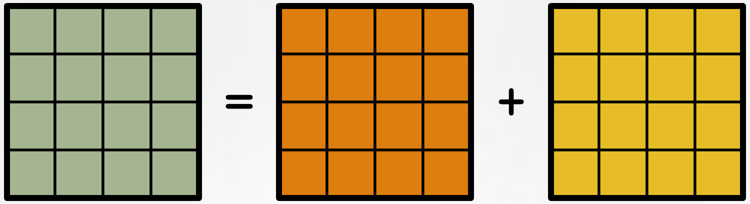



```
var A, B: [1..n] real, key: real;

var (val, loc) = maxloc reduce (A==key, 1..n);
if val == false then
  loc = n
else
  loc = loc - 1;

forall i in 1..loc do
  A(i) = B(i);
```

The Many Ways of Writing A = B + C



1. Whole array assignment and promotion of ‘+’

```
A = B + C;
```

2. Forall loop over arrays

```
forall (a,b,c) in (A,B,C) do
  a = b + c;
```

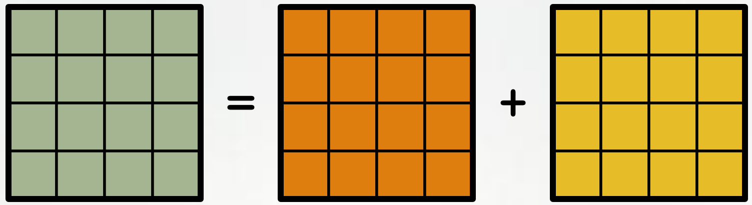
1. Forall loop over domain (assumes arrays’ domains equal D)

```
forall i in D do
  A(i) = B(i) + C(i);
```

2. Forall loop over arrays’ domains

```
forall (i,j,k) in (A.domain, B.domain, C.domain) do
  A(i) = B(j) + C(k);
```

The Many Ways of Writing A = B + C



5. Whole array assignment and promotion of '+' of slices

```
A(D) = B(D) + C(D);
```

6. Forall loop over array slices

```
forall (a,b,c) in (A(D), B(D), C(D)) do
  a = b + c;
```

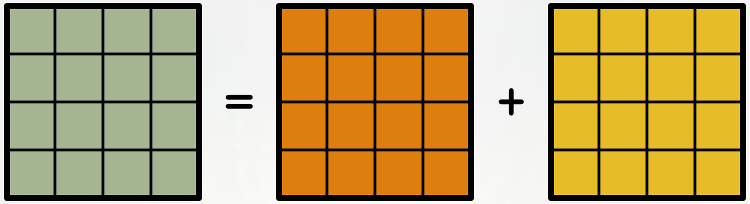
7. Whole array assignment and promotion of slices with ranges

```
A[.., D.low(2)..D.high(2)] =
  B[..D.high(1), D.low(2)..] + C[D.dims(1), ..];
```

8. Forall loop over A's domain with re-indexed aliases of B and C

```
var BB: [A.domain] => B, CC: [A.domain] => C;
[i in A.domain] A(i) = BB(i) + CC(i);
```

The Many Ways of Writing A = B + C



9. Whole array assignment of a forall expressions

```
A = forall (b,c) in (B,C) do b+c;
```

10. Forall loop over an array and the promotion of '+'

```
forall (a,bc) in (A, B+C) do
  a = bc;
```

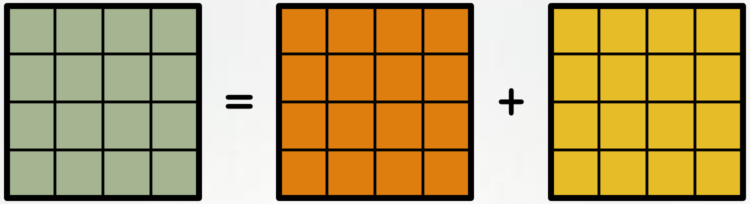
11. Assignment to a forall expression from promoted expressions

```
([a in A] a) =
  ([b in B] b) + [c in C] c;
```

12. Forall loop over arrays and a forall expression

```
forall (a,b,c) in (A, B, forall c in C do c) do
  a = b + c;
```

The Many Ways of Writing A = B + C



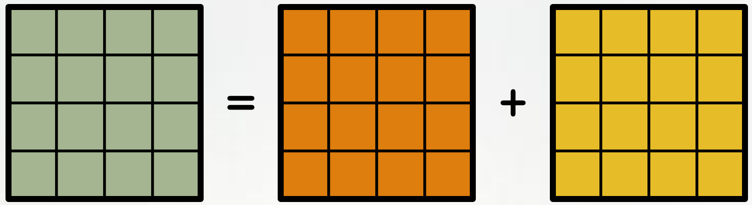
13. Task-parallel loop with data-parallel array slice assignments

```
coforall t in 1..numTasks do
  A(chunk(D,t)) = B(chunk(D,t)) + C(chunk(D,t));
```

- Similar performance is ultimately expected for all variations
- Compiler translates code using array implementations
  - All arrays are implemented in Chapel (Distributions and Layouts)
  - Compiler targets a structural interface in Chapel

Note: These examples are overkill for A=B+C, but they demonstrate the orthogonality and power of Chapel's data-parallel abstractions.

Compiler Translation of A = B + C



Promotion of ‘+’ produces a forall expression

```
A = forall (b,c) in (B, C) do b+c;
```

Array assignment is implemented via a zippered forall loop

```
forall (a,(b,c)) in (A, forall (b,c) in (B,C) do b+c) do
  a = b + c;
```

Parallel iteration over a default array uses a coforall

```
coforall t in 1..numTasks do
  forall (a,(b,c)) in
    (A(chunk(D,t)),
     [(b,c) in (B(chunk(D,t)),C(chunk(D,t)))] b+c) do
    a = b + c;
```

Questions?

- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
  - Promotion
  - Reductions
  - Scans
- Examples
  - NAS MG stencil revisited
  - Pre-key copy kernel
  - A = B + C