Chapel: Data Parallelism

Brad Chamberlain

PRACE Winter School
12 February 2009

Outline

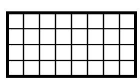
- Domains and Arrays
  - overview
  - arithmetic domains
  - domain roles
- Other Domain Types
- Data Parallel Operations
- Example Computations

Domains

- *domain:* a first-class index set
  - specifies size and shape of arrays
  - supports iteration, array operations
  - potentially distributed across machine
- Three main classes:
  - *arithmetic:* indices are Cartesian tuples
    - rectilinear, multidimensional
    - optionally strided and/or sparse
  - *associative:* indices serve as hash keys
    - supports hash tables, dictionaries
  - *opaque:* indices are anonymous
    - supports sets, graph-based computations
- Fundamental Chapel concept for data parallelism
- A generalization of ZPL's *region* concept

Sample Arithmetic Domains

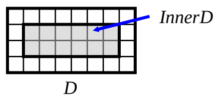
```
var m = 4, n = 8;

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



D

Sample Arithmetic Domains

```
var m = 4, n = 8;

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



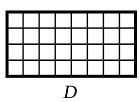
Domain Roles: Declaring Arrays

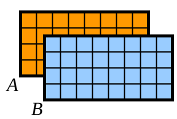
- Syntax

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

- Semantics
  - for each index in *domain-expr*, stores an element of *type*
- Example

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





- Revisiting our previous array declarations:

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

Domain Roles: Supporting Iteration

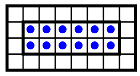
- Syntax

```
for    index-expr in domain-expr loop-body
forall index-expr in domain-expr loop-body
```

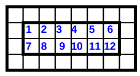
- Semantics
  - *for* – same as previous for-loops we've seen; indices are const
  - *forall* – asserts the loop iterations can/should be executed in parallel
    – also that they are serializable (can be run by a single task)
- Example

```
forall (i,j) in InnerD do
  A(i,j) = i + j/10.0;
for ind in InnerD { write(A(ind), " "); }
```

D



D



- Output

```
2.2 2.3 2.4 2.5 2.6 2.7 3.2 3.3 3.4 3.5 3.6 3.7
```


Other forall loop forms

- Forall loops also support…

…an expression-based form:

```
var A: [D] real = forall (i,j) in D do i + j/10.0;
```

…a symbolic shorthand:

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

…and a sugar that combines it with the array type declaration syntax:

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

// can be written:

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

Loops and Parallelism

- *for loops:* one task executes all iterations
- *forall loops:* some number of tasks executes the iterations
  - as determined by the iterator expression controlling the loop
    - for domains/arrays, specified as part of its *distribution*
    - for other objects/iterators, author specifies using task parallelism
- *coforall loops:* one task per iteration

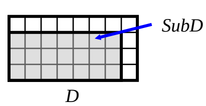
Domain Roles: Domain Slicing

- Syntax

```
domain-slice:
  domain-expr[domain-expr]
```

- Semantics
  - evaluates to the intersection of the two domains
- Example

```
const SubD: subdomain(D) = D[2.., ..7];
```



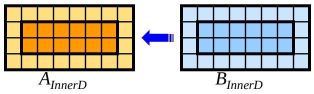
Domain Roles: Array Slicing

- Syntax

```
array-slice:
  array-expr[domain-expr]
```

- Semantics
  - evaluates to the sub-array referenced by *domain-expr*
  - *domain-expr*'s indices must be legal for *array-expr*
- Example

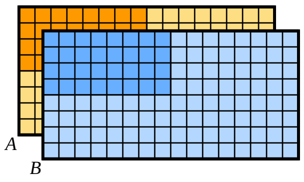
```
A[InnerD] = B[InnerD];
```



Domain Roles: Array Reallocation

- Semantics
  - re-assigning a domain's index set causes its arrays to be reallocated
  - array values are preserved for indices that remain in the index set
  - elements for new indices are initialized to the type's default value
- Example

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



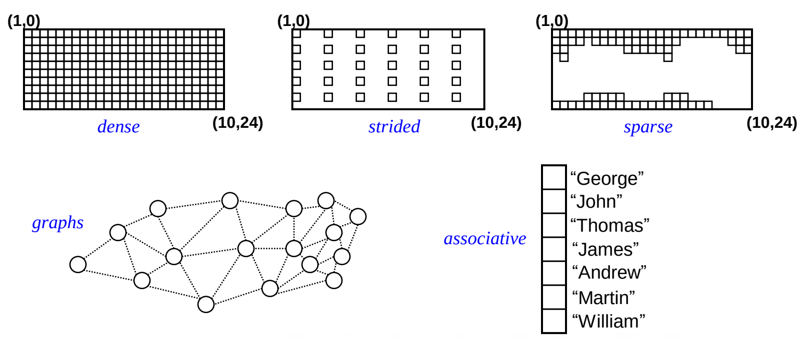
Outline

- Domains
- Other Domain Types
  - strided
  - sparse
  - associative
  - opaque
- Data Parallel Operations
- Example Computations

Other Domain Types

Domain indices can be…

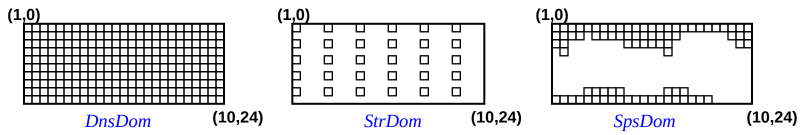
…integer tuples…

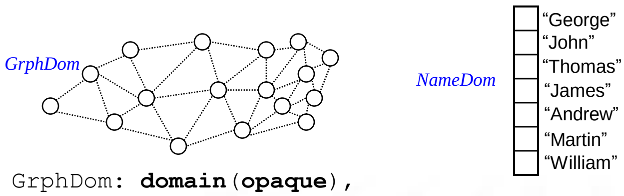


…anonymous… …or arbitrary values.

Domain Declarations

```
var DnsDom: domain(2) = [1..10, 0..24],
    StrDom: subdomain(DnsDom) = DnsDom by (2,4),
    SpsDom: subdomain(DnsDom) = genIndices();
```

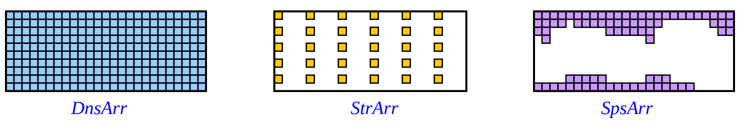


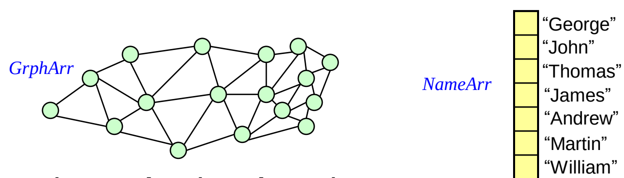



```
var GrphDom: domain(opaque),
    NameDom: domain(string) = readNames();
```


Array Declarations

```
var DnsArr: [DnsDom] complex,
    StrArr: [StrDom] real(32),
    SpsArr: [SpsDom] real;
```



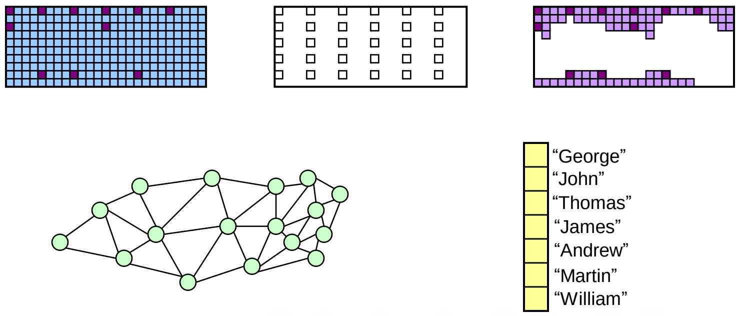



```
var GrphArr: [GrphDom] string,
    NameArr: [NameDom] int(64);
```

Data Parallelism: Domain Iteration

All domain types support iteration…

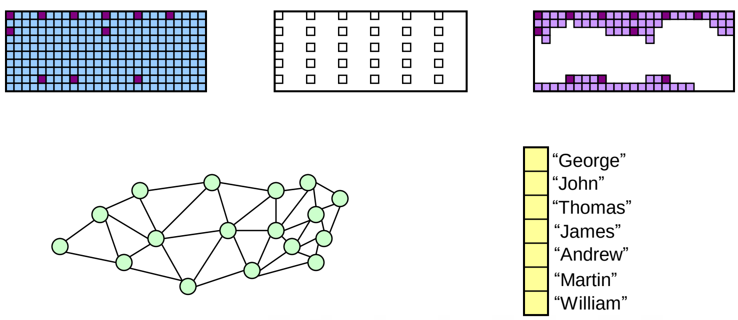
```
forall ij in StrDom {
  DnsArr(ij) += SpsArr(ij);
}
```



Data Parallelism: Array Slicing

…array slicing…

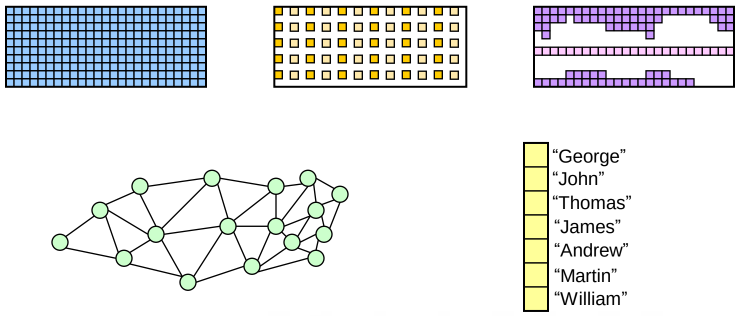
```
DnsArr[StrDom] += SpsArr[StrDom];
```



Data Parallelism: Array Reallocation

…array reallocation (and all other domain/array operations)

```
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
```



The Domain/Index Hierarchy

domain(1) domain(2) domain(3) … domain(opaque)

D DnsDom

InnerD StrDom SpsDom

ij implicitly defined as:
`const ij: index(InnerD);`

```
forall ij in InnerD { …A(ij)… }
```

No bounds check needed since index(InnerD) ∊ InnerD ⊂ D == domain(A)

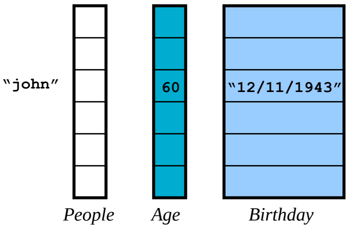
index(1) index(2) index(3) … index(opaque)

index(D) index(D2)

index(InnerD) index(StrDom) index(SpsDom)

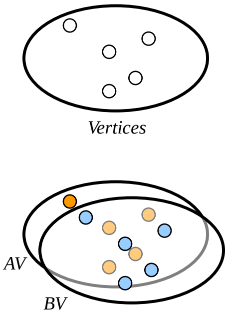
Associative Domains and Arrays

```
var People: domain(string);
var Age: [People] int,
    Birthdate: [People] string;


People += "john";
Age("john") = 60;
Birthdate("john") = "12/11/1943";

…
forall person in People {
  if (Birthdate(person) == today) {
    Age(person) += 1;
  }
}
```



Opaque Domains and Arrays

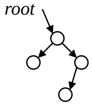
```
var Vertices: domain(opaque);

for i in (1..5) {
  Vertices.create();
}


var AV, BV: [Vertices] real;
```



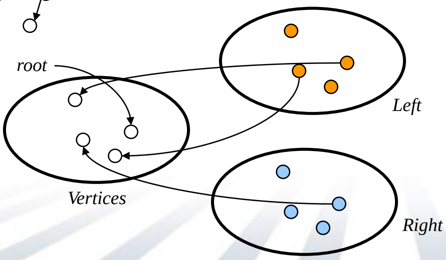
Opaque Domains and Arrays

```
var Vertices: domain(opaque);
var left, right: [Vertices] index(Vertices);
var root: index(Vertices);

root = Vertices.create();
left(root) = Vertices.create();
right(root) = Vertices.create();
left(right(root)) = Vertices.create();
```

conceptually:



more precisely:



Outline

- Domains
- Other Domain Types
- Data Parallel Operations
  - promotion
  - reductions and scans
- Example Computations

Promotion

- Functions/operators expecting scalar values can also take…

  …arrays, causing each element to be passed in

```
A = sin(B);
B = 2 * A;
```

  …domains, causing each index to be passed in

```
def foo(x: (int, int)) { … }
foo(SpsDom);        // calls foo once per index in SpsDom
```

- When multiple arguments are promoted, calls may use…

  …zippered promotion:

```
X = pow(A, B); // X is 2D; X(i,j) = pow(A(i,j), B(i,j))
```

  …tensor promotion:

```
Y = pow[A, B]; // Y is 2x2D;
               // Y(i,j)(k,l) = pow(A(i,j), B(k,l))
```


Promotion and Parallelism

- Promoted functions/operators are executed in parallel
  - as if a forall loop implements the calls using zipper/tensor iteration

```
A = sin(B);
B = 2 * A;

foo(SpsDom);

X = pow(A, B);

X = pow[A, B];
```

≈

```
A = [b in B] sin(b);
B = [a in A] 2 * a;

[i in SpsDom] foo(i);

X = [(a,b) in (A,B)] pow(a,b);

X = [(a,b) in [A,B]] pow(a,b);
```

Reductions

- Syntax

```
reduce-expr:
  reduce-type reduce iteratable-expr
```

- Semantics
  - combines elements generated by *iteratable-expr* using *reduce-type*
  - *reduce-type* may be one of several built-in operators, or user-defined
- Examples

```
tot = + reduce A;  // tot is the sum of all elements in A
big = max reduce [i in InnerD] abs(A(i) + B(i));
```

- Future work:
  - support for partial reductions to reduce only a subset of an array's dimensions

Scans

- Syntax

```
scan-expr:
  scan-type scan iteratable-expr
```

- Semantics
  - combines elements generated by *iteratable-expr* using *scan-type,* generating partial results along the way
  - *scan-type* may be one of several built-in operators, or user-defined
- Examples

```
var A, B, C: [1..5] real;
A = 1.1;                 // A is: 1.1 1.1 1.1 1.1 1.1
B = + scan A;            // B is: 1.1 2.2 3.3 4.4 5.5
B(3) = -B(3);            // B is: 1.1 2.2 -3.3 4.4 5.5
C = min scan B;          // C is: 1.1 1.1 -3.3 -3.3 -3.3
```

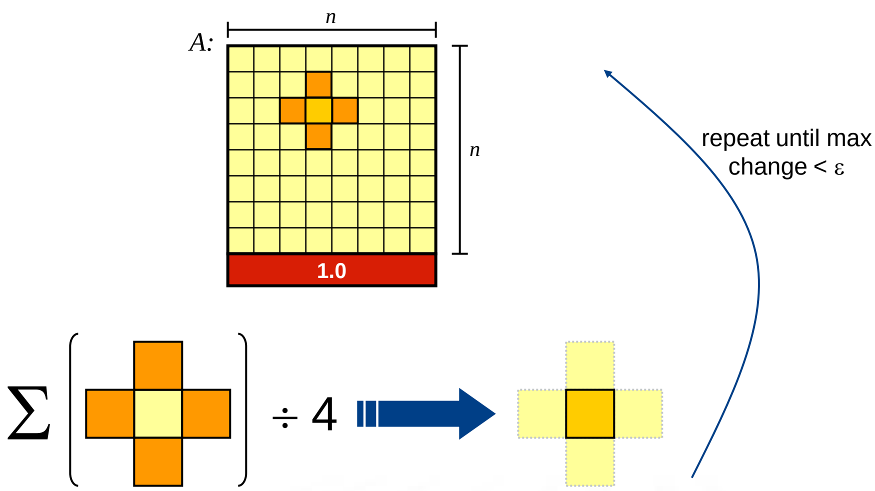
Reduction/Scan operators

- Built-in:
  - `+`, `-`, `*`, `/`, `&`, `|`, `^`, `&&`, `||`, **`min`**, **`max`**: do the obvious things
  - **`minloc`**, **`maxloc`**: generate a tuple result: (min/max value, its index)
- User-defined:
  - user must define a class that supports a number of methods to:
    - generate a new identity state value
    - combine the state element with a new element
    - combine two state elements
    - generate an output result
    - …
  - the compiler generates a code template to compute the operation in parallel, utilizing the user's class methods
  - for more information, see:
    - 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 Practice of Parallel Programming, 2006.

Outline

- Domains
- Other Domain Types
- Data Parallel Operations
- Example Computations
  - Jacobi iteration
  - Multigrid

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 arguments

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 ⇒ **integer** (current default, 32 bits)

epsilon ⇒ **floating-point** (current default, 64 bits)

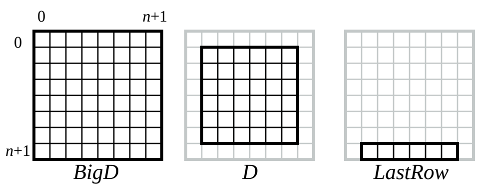
Jacobi Iteration in Chapel

```
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);
```

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



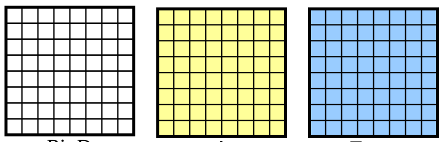
exterior ⇒ one of several built-in domain generators

Jacobi Iteration in Chapel

```
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;
```

Declare arrays

var ⇒ can be modified throughout its lifetime
: *T* ⇒ declares variable to be of type *T*
: *[D] T* ⇒ array of size *D* with elements of type *T*
(no initializer) ⇒ values initialized to default value (0.0 for reals)



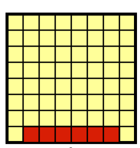
BigD *A* *Temp*

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



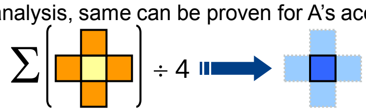
A

Jacobi Iteration in Chapel

Compute 5-point stencil

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

Note: since (*i,j*) ∈ *D* and *D* ⊆ *BigD* and *Temp*: [*BigD*]
⇒ no bounds check required for *Temp(i,j)*
with compiler analysis, same can be proven for A's accesses




```
  [(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);
```

Copy data back & Repeat until done

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 {
  [(                                          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

If written to a file, parallel I/O would be used

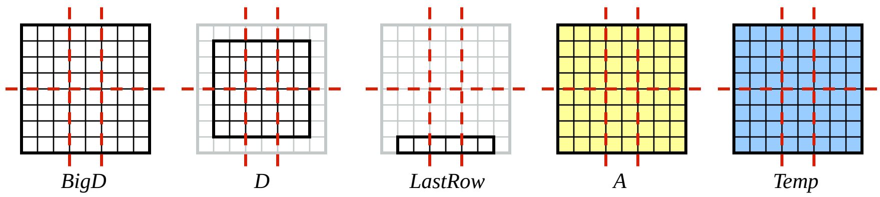
Jacobi Iteration in Chapel

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

const BigD: domain(2) = [0..n+1, 0..n+1] distributed 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 location of iterations over locales

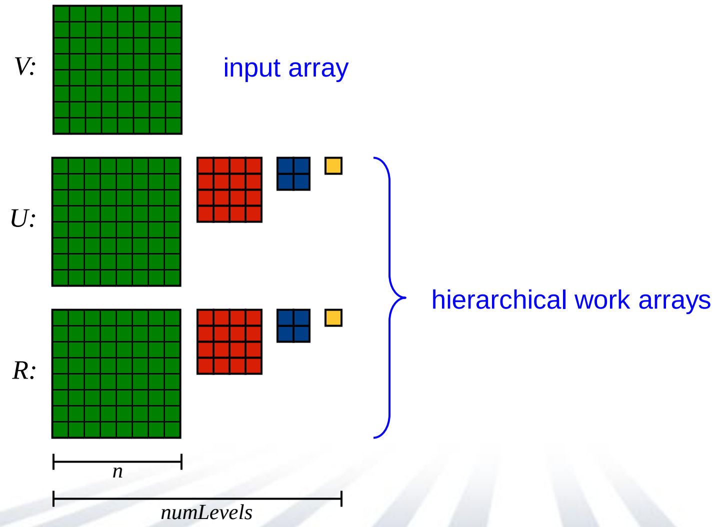
Subdomains inherit parent domain's distribution



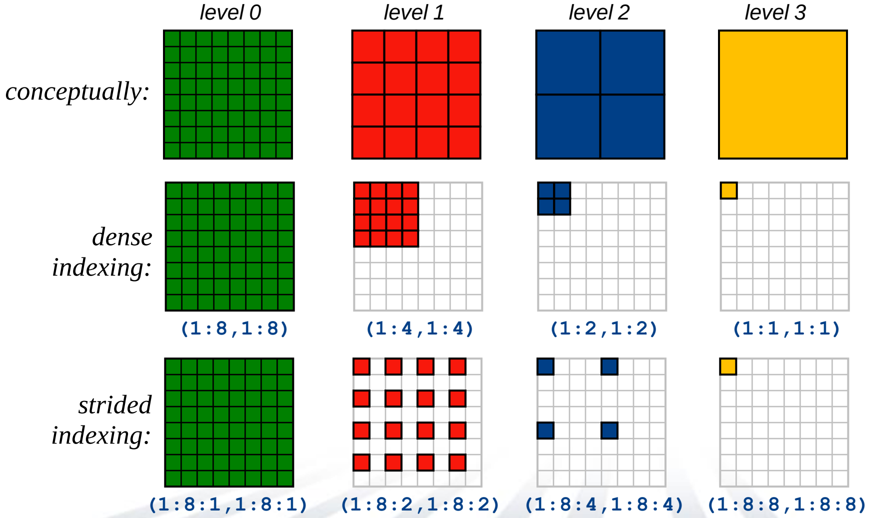
Jacobi Iteration in Chapel

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

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

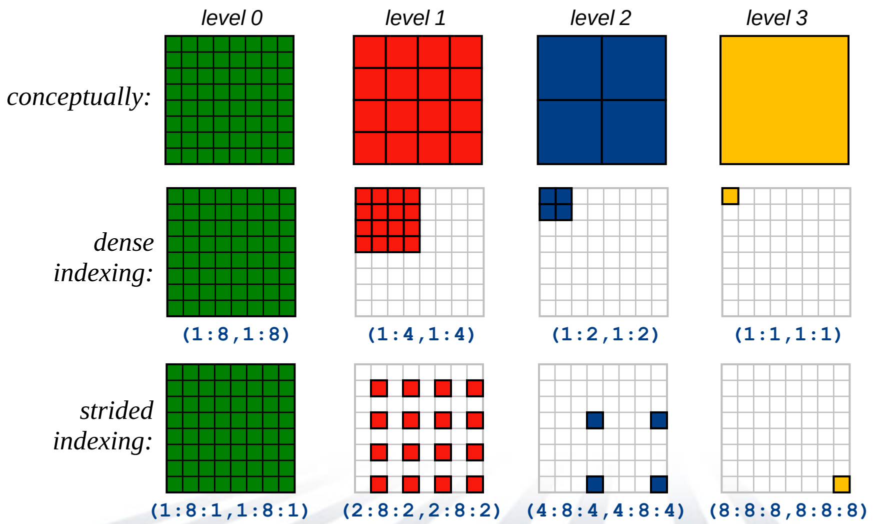
Multigrid Example



Hierarchical Arrays



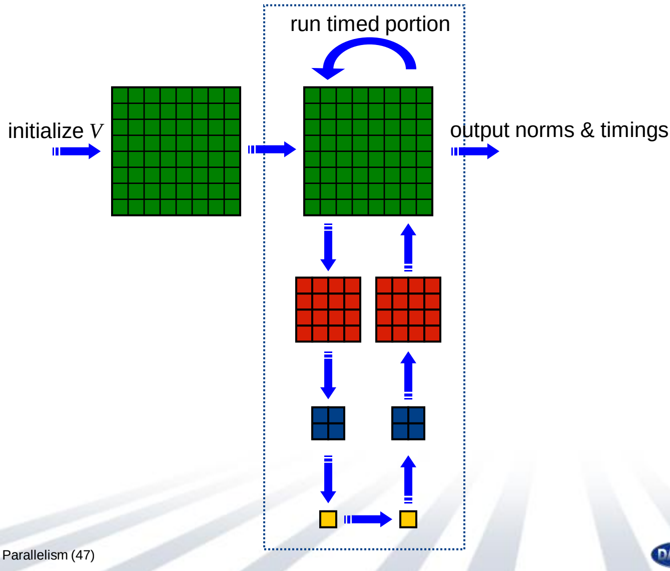
Hierarchical Arrays



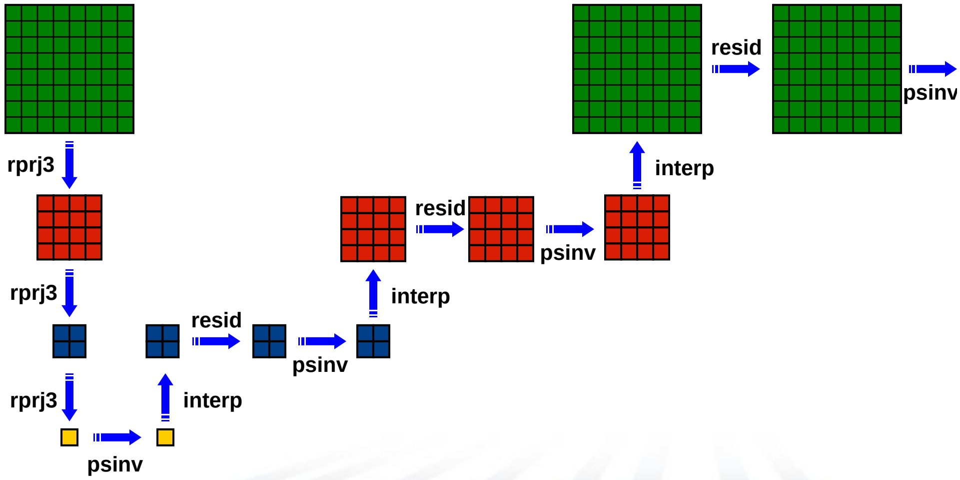
Hierarchical Array Declarations in Chapel

```
config const n = 1024,
             numLevels = lg2(n);

const Levels = [0..#numLevels];
const ProblemSpace: domain(3) distributed BlockWrap
                  = [1..n, 1..n, 1..n];

var V: [ProblemSpace] real;

const HierSpace: [lvl in Levels] subdomain(ProblemSpace)
               = ProblemSpace by -2**lvl;

var U, R: [lvl in Levels] [HierSpace(lvl)] real;
```

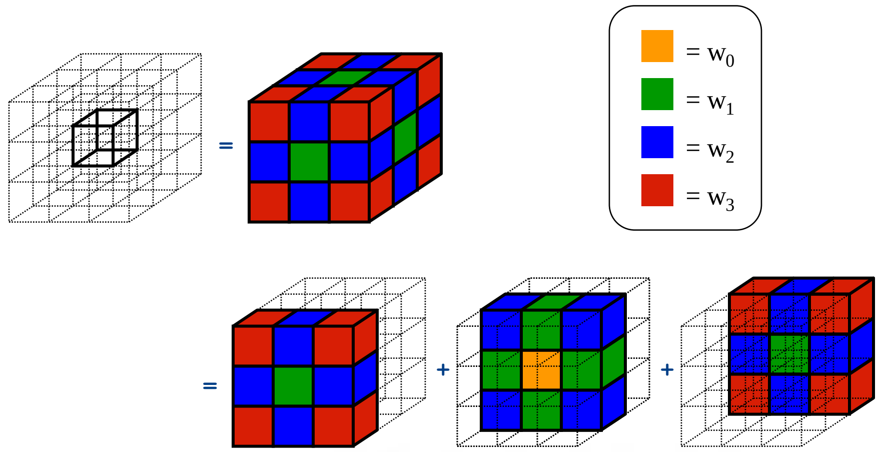
Overview of NAS MG



MG's projection/interpolation cycle



NAS MG: *rprj3* stencil



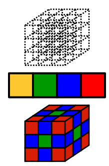
Multigrid: Stencils in Chapel

- Can write them out explicitly, as in Jacobi…

```
def rprj3(S, R) {
  const w: [0..3] real = (0.5, 0.25, 0.125, 0.0625);
  const Rstr = R.stride;

  forall ijk in S.domain do
    S(ijk) = w(0) * R(ijk)
           + w(1) * (R(ijk+Rstr*(1,0,0)) + R(ijk+Rstr*(-1,0,0))
                   + R(ijk+Rstr*(0,1,0)) + R(ijk+Rstr*(0,-1,0))
                   + R(ijk+Rstr*(0,0,1)) + R(ijk+Rstr*(0,0,-1)))
           + w(2) * (R(ijk+Rstr*(1,1,0)) + R(ijk+Rstr*(1,-1,0))
                   + R(ijk+Rstr*(-1,1,0)) + R(ijk+Rstr*(-1,-1,0))
                   + R(ijk+Rstr*(1,0,1)) + R(ijk+Rstr*(1,0,-1))
                   + R(ijk+Rstr*(-1,0,1)) + R(ijk+Rstr*(-1,0,-1))
                   + R(ijk+Rstr*(0,1,1)) + R(ijk+Rstr*(0,1,-1))
                   + R(ijk+Rstr*(0,-1,1)) + R(ijk+Rstr*(0,-1,-1)))
           + w(3) * (R(ijk+Rstr*(1,1,1) + R(ijk+Rstr*(1,1,-1))
                   + R(ijk+Rstr*(1,-1,1) + R(ijk+Rstr*(1,-1,-1))
                   + R(ijk+Rstr*(-1,1,1) + R(ijk+Rstr*(-1,1,-1))
                   + R(ijk+Rstr*(-1,-1,1) + R(ijk+Rstr*(-1,-1,-1)));
}
```

Multigrid: Stencils in Chapel

- …or, note that a stencil is simply a reduction over a small subarray expression
- Thus, stencils can be written in a “syntactically scalable” way using reductions:



```
def rprj3(S, R) {
  const Stencil: domain(3) = [-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 S.domain do
    S(ijk) = + reduce [off in Stencil]
                         (w3d(off) * R(ijk + R.stride*off));
}
```


NAS MG *rprj3* stencil in Fortran+MPI

```
      subroutine comm3(u,n1,n2,n3,kk)
      use caf_intrinsics

      implicit none

      include 'cafnpb.h'
      include 'globals.h'

      integer n1, n2, n3, kk
      double precision u(n1,n2,n3)
      integer axis

      if( .not. dead(kk) )then
         do  axis = 1, 3
            if( nprocs .ne. 1) then
               call sync_all()
               call give3( axis, +1, u, n1, n2, n3, kk )
               call give3( axis, -1, u, n1, n2, n3, kk )
               call sync_all()
               call take3( axis, -1, u, n1, n2, n3 )
               call take3( axis, +1, u, n1, n2, n3 )
            else
               call comm1p( axis, u, n1, n2, n3, kk )
            endif
         enddo
      else
         do  axis = 1, 3
            call sync_all()
            call sync_all()
         enddo
         call zero3(u,n1,n2,n3)
      endif
      return
      end


      subroutine give3( axis, dir, u, n1, n2, n3, k )
      use caf_intrinsics


      implicit none

      include 'cafnpb.h'
      include 'globals.h'

      integer axis, dir, n1, n2, n3, k, ierr
      double precision u( n1, n2, n3 )

      integer i3, i2, i1, buff_len,buff_id

      buff_id = 2 + dir
      buff_len = 0

      if( axis .eq.  1 )then
         if( dir .eq. -1 )then

            do  i3=2,n3-1
               do  i2=2,n2-1
                  buff_len = buff_len + 1
                  buff(buff_len,buff_id ) = u( 2,  i2,i3)
               enddo
            enddo

            buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
     >      buff(1:buff_len,buff_id)

         else if( dir .eq. +1 ) then

            do  i3=2,n3-1
               do  i2=2,n2-1
                  buff_len = buff_len + 1
                  buff(buff_len, buff_id ) = u( n1-1, i2,i3)
               enddo
            enddo

            buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
     >      buff(1:buff_len,buff_id)

         endif
      endif

      if( axis .eq.  2 )then
         if( dir .eq. -1 )then
            do  i3=2,n3-1
               do  i1=1,n1
                  buff_len = buff_len + 1
                  buff(buff_len, buff_id ) = u( i1,  2,i3)
               enddo
            enddo

       buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
     >      buff(1:buff_len,buff_id)

         else if( dir .eq. +1 ) then

            do  i3=2,n3-1
               do  i1=1,n1
                  buff_len = buff_len + 1
                  buff(buff_len,  buff_id )= u( i1,n2-1,i3)
               enddo
            enddo

            buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
     >      buff(1:buff_len,buff_id)

         endif
      endif

      if( axis .eq.  3 )then
         if( dir .eq. -1 )then

            do  i2=1,n2
               do  i1=1,n1
                  buff_len = buff_len + 1
                  buff(buff_len, buff_id ) = u( i1,i2,2)
               enddo
            enddo

            buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
     >      buff(1:buff_len,buff_id)

         else if( dir .eq. +1 ) then

            do  i2=1,n2
               do  i1=1,n1
                  buff_len = buff_len + 1
                  buff(buff_len, buff_id ) = u( i1,i2,n3-1)
               enddo
            enddo

            buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
     >      buff(1:buff_len,buff_id)

         endif
      endif

      return
      end


      subroutine take3( axis, dir, u, n1, n2, n3 )
      use caf_intrinsics


      implicit none

      include 'cafnpb.h'
      include 'globals.h'

      integer axis, dir, n1, n2, n3
      double precision u( n1, n2, n3 )

      integer buff_id, indx

      integer i3, i2, i1

      buff_id = 3 + dir
      indx = 0

      if( axis .eq.  1 )then
         if( dir .eq. -1 )then

            do  i3=2,n3-1
               do  i2=2,n2-1
                  indx = indx + 1
                  u(n1,i2,i3) = buff(indx, buff_id )
               enddo
            enddo

         else if( dir .eq. +1 ) then

            do  i3=2,n3-1
               do  i2=2,n2-1
                  indx = indx + 1
                  u(1,i2,i3) = buff(indx, buff_id )
               enddo
            enddo

         endif
      endif

      if( axis .eq.  2 )then
         if( dir .eq. -1 )then

            do  i3=2,n3-1
               do  i1=1,n1
                  indx = indx + 1
                  u(i1,n2,i3) = buff(indx, buff_id )
               enddo
            enddo

         else if( dir .eq. +1 ) then

            do  i3=2,n3-1
               do  i1=1,n1
                  indx = indx + 1
                  u(i1,1,i3) = buff(indx, buff_id )
               enddo
            enddo

         endif
      endif

      if( axis .eq.  3 )then
         if( dir .eq. -1 )then

            do  i2=1,n2
               do  i1=1,n1
                  indx = indx + 1
                  u(i1,i2,n3) = buff(indx, buff_id )
               enddo
            enddo

         else if( dir .eq. +1 ) then

            do  i2=1,n2
               do  i1=1,n1
                  indx = indx + 1
                  u(i1,i2,1) = buff(indx, buff_id )
               enddo
            enddo

         endif
      endif

      return
      end


      subroutine comm1p( axis, u, n1, n2, n3, kk )
      use caf_intrinsics

      implicit none

      include 'cafnpb.h'
      include 'globals.h'

      integer axis, dir, n1, n2, n3
      double precision u( n1, n2, n3 )

      integer i3, i2, i1, buff_len,buff_id
      integer i, kk, indx

      dir = -1

      buff_id = 3 + dir
      buff_len = nm2

      do  i=1,nm2
         buff(i,buff_id) = 0.0D0
      enddo


      dir = +1

      buff_id = 3 + dir
      buff_len = nm2

      do  i=1,nm2
         buff(i,buff_id) = 0.0D0
      enddo

      dir = +1

      buff_id = 2 + dir
      buff_len = 0

      if( axis .eq.  1 )then
         do  i3=2,n3-1
            do  i2=2,n2-1
               buff_len = buff_len + 1
               buff(buff_len, buff_id ) = u( n1-1,
       i2,i3)
            enddo
         enddo
      endif

      if( axis .eq.  2 )then
         do  i3=2,n3-1
            do  i1=1,n1
               buff_len = buff_len + 1
               buff(buff_len,  buff_id )= u( i1,n2-
       1,i3)
            enddo
         enddo
      endif

      if( axis .eq.  3 )then
         do  i2=1,n2
            do  i1=1,n1
               buff_len = buff_len + 1
               buff(buff_len, buff_id ) = u( i1,i2,n3-
       1)
            enddo
         enddo
      endif

      dir = -1

      buff_id = 2 + dir
      buff_len = 0

      if( axis .eq.  1 )then
         do  i3=2,n3-1
            do  i2=2,n2-1
               buff_len = buff_len + 1
               buff(buff_len,buff_id ) = u( 2,  i2,i3)
            enddo
         enddo
      endif

      if( axis .eq.  2 )then
         do  i3=2,n3-1
            do  i1=1,n1
               buff_len = buff_len + 1
               buff(buff_len, buff_id ) = u( i1,
       2,i3)
            enddo
         enddo
      endif

      if( axis .eq.  3 )then
         do  i2=1,n2
            do  i1=1,n1
               buff_len = buff_len + 1
               buff(buff_len, buff_id ) = u( i1,i2,2)
            enddo
         enddo
      endif

      do  i=1,nm2
         buff(i,4) = buff(i,3)
         buff(i,2) = buff(i,1)
      enddo

      dir = -1

      buff_id = 3 + dir
      indx = 0

      if( axis .eq.  1 )then
         do  i3=2,n3-1
            do  i2=2,n2-1
               indx = indx + 1
               u(n1,i2,i3) = buff(indx, buff_id )
            enddo
         enddo
      endif

      if( axis .eq.  2 )then
         do  i3=2,n3-1
            do  i1=1,n1
               indx = indx + 1
               u(i1,n2,i3) = buff(indx, buff_id )
            enddo
         enddo
      endif

      if( axis .eq.  3 )then
         do  i2=1,n2
            do  i1=1,n1
               indx = indx + 1
               u(i1,i2,n3) = buff(indx, buff_id )
            enddo
         enddo
      endif


      dir = +1

      buff_id = 3 + dir
      indx = 0

      if( axis .eq.  1 )then
         do  i3=2,n3-1
            do  i2=2,n2-1
               indx = indx + 1
               u(1,i2,i3) = buff(indx, buff_id )
            enddo
         enddo
      endif

      if( axis .eq.  2 )then
         do  i3=2,n3-1
            do  i1=1,n1
               indx = indx + 1
               u(i1,1,i3) = buff(indx, buff_id )
            enddo
         enddo
      endif

      if( axis .eq.  3 )then
         do  i2=1,n2
            do  i1=1,n1
               indx = indx + 1
               u(i1,i2,1) = buff(indx, buff_id )
            enddo
         enddo
      endif

      return
      end

      subroutine rprj3(r,m1k,m2k,m3k,s,m1j,m2j,m3j,k)
      implicit none
      include 'cafnpb.h'
      include 'globals.h'

      integer m1k, m2k, m3k, m1j, m2j, m3j,k

      double precision r(m1k,m2k,m3k), s(m1j,m2j,m3j)
      integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
      double precision x1(m), y1(m), x2,y2

      if(m1k.eq.3)then
        d1 = 2
      else
        d1 = 1
      endif

      if(m2k.eq.3)then
        d2 = 2
      else
        d2 = 1
      endif

      if(m3k.eq.3)then
        d3 = 2
      else
        d3 = 1
      endif

      do  j3=2,m3j-1
        i3 = 2*j3-d3
        do  j2=2,m2j-1
          i2 = 2*j2-d2
          do j1=2,m1j
            i1 = 2*j1-d1
            x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
     >               + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
            y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
     >               + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
          enddo
          do  j1=2,m1j-1
            i1 = 2*j1-d1
            y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
     >         + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1)
            x2 = r(i1,  i2-1,i3  ) + r(i1,  i2+1,i3  )
     >         + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
            s(j1,j2,j3) =
     >          0.5D0 * r(i1,i2,i3)
     >        + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2)
     >        + 0.125D0 * ( x1(i1-1) + x1(i1+1) + y2)
     >        + 0.0625D0 * ( y1(i1-1) + y1(i1+1) )
          enddo
        enddo
      enddo
      j = k-1
      call comm3(s,m1j,m2j,m3j,j)
      return
      end
```

Data Parallelism Status

- Stable Features:
  - most features in this section are implemented, but using a single task
- Incomplete Features:
  - forall loops, promotion, and reductions do not result in parallelism yet
  - promoted functions do not preserve array shape by default
  - index types and subdomains are not bounds-checked
- Unimplemented Features:
  - arrays of differently-sized arrays are not yet supported
  - partial reductions and scans are not yet defined or implemented
  - user defined reductions and scans are not yet supported

Questions?