Chapel: Data Parallelism

Brad Chamberlain

SC08: Tutorial S07 – 11/16/08

Outline

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

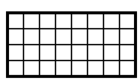
Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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];
```

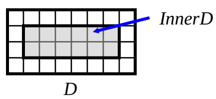


D

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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];
```



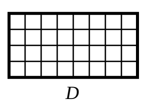
Domain Roles: Declaring Arrays

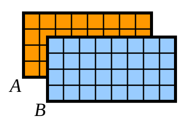
- Syntax

```
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]
```


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

CRAY

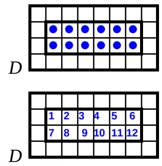
Domain Roles: Supporting Iteration

- Syntax

```
for identifier in domain-expr loop-body
forall identifier 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), " "); }
```



- 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
```


DARPA HPCS

CRAY

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


DARPA HPCS

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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

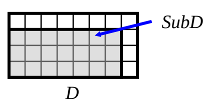
Domain Roles: Domain Slicing

- Syntax

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

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

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



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

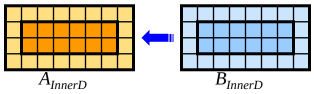
Domain Roles: Array Slicing

- Syntax

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

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

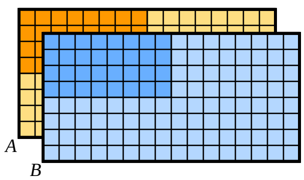
```
A[InnerD] = B[InnerD];
```



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];
```



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

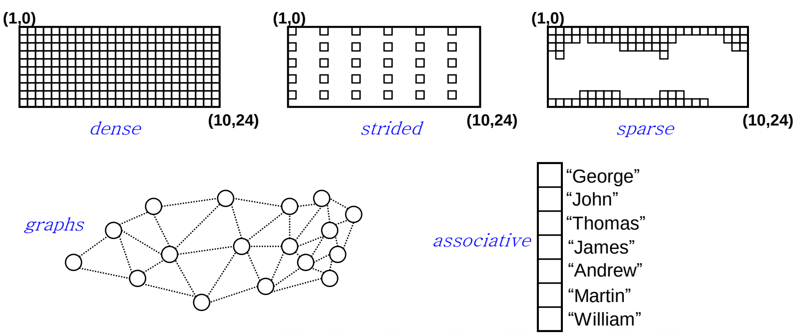
Outline

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

Other Domain Types

Domain indices can be…

…integer tuples…

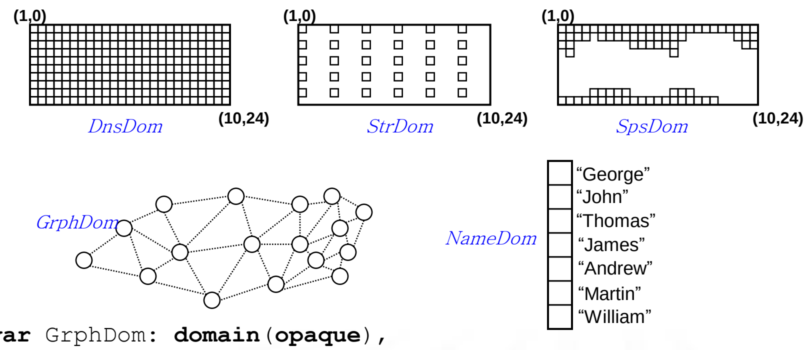


…anonymous… …or arbitrary values.

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

Domain Declarations

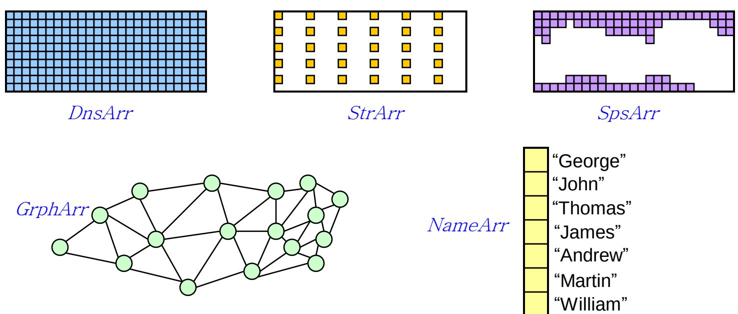
```
var DnsDom: domain(2) = [1..10, 0..24],
    StrDom: subdomain(DnsDom) = DnsDom by (2,4),
    SpsDom: subdomain(DnsDom) = genIndices();
```




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


Array Declarations

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



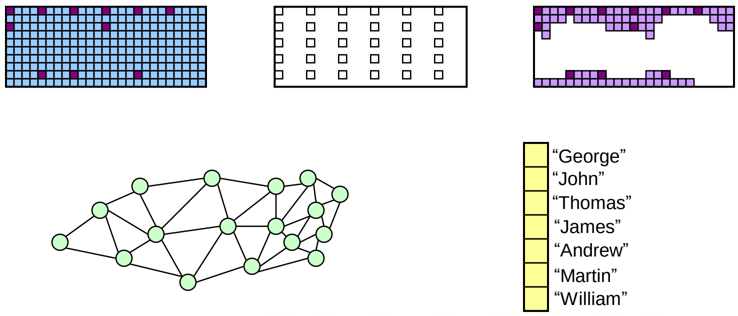

```
var GrphArr: [GrphDom] string,
    NameArr: [NameDom] int(64);
```


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

Data Parallelism: Domain Iteration

All domain types support iteration…

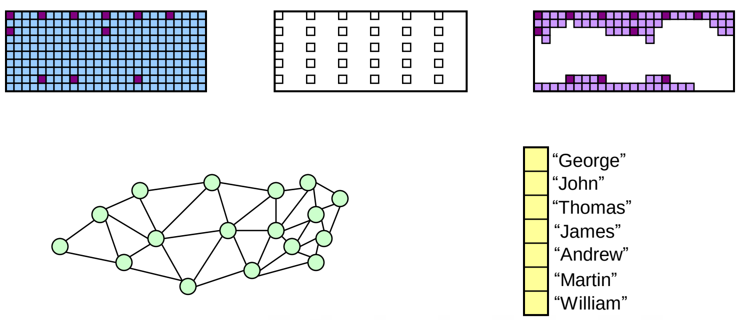
```
forall ij in StrDom {
  DnsArr(ij) += SpsArr(ij);
}
```



Data Parallelism: Array Slicing

…array slicing…

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

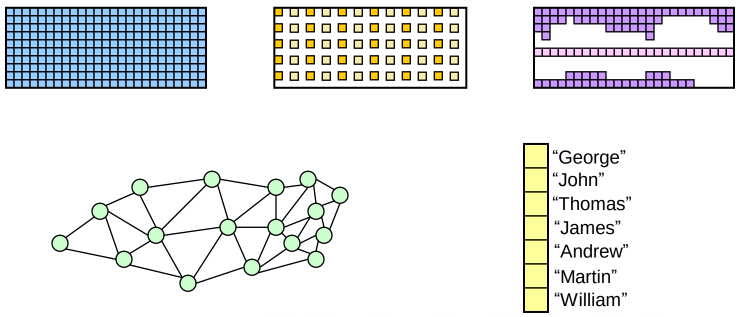


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

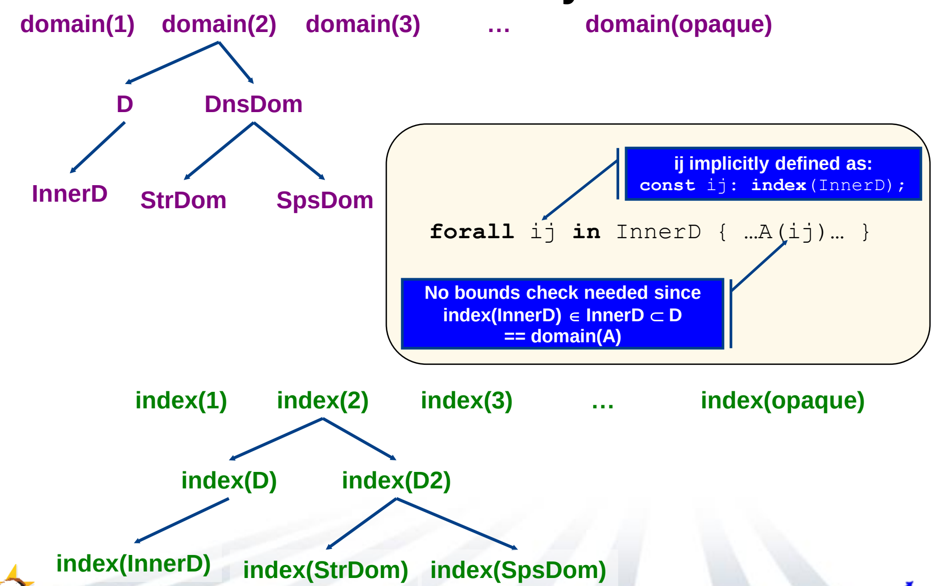
Data Parallelism: Array Reallocation

…array reallocation (as well as other domain/array operations)

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



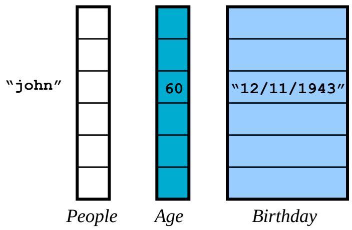
The Domain/Index Hierarchy



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

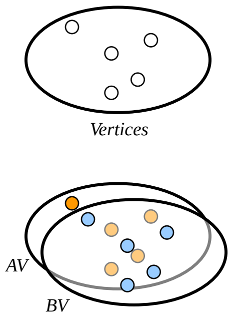
Associative Domains and Arrays

```
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

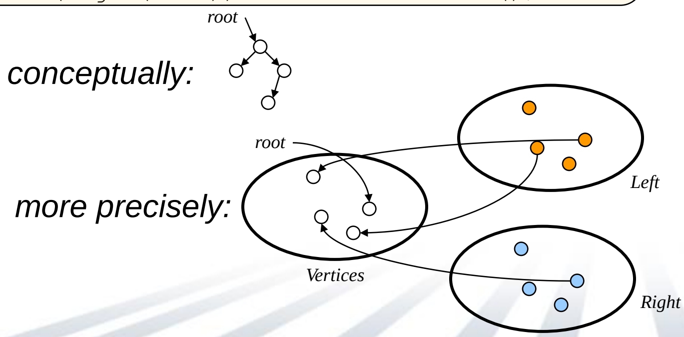
```
var Vertices: domain(opaque);

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



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



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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();
```



Outline

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

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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 zipper/tensor forall loop was implementing the promotion

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

foo(SpsDom);          ≈     [i in SpsDom] foo(i);

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

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


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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

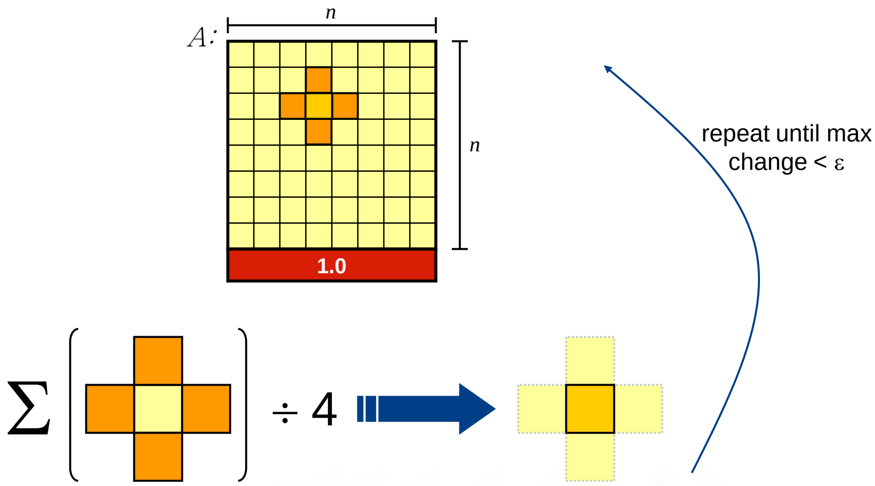
- Built-in:
  - `+`, `-`, `*`, `/`, `&`, `|`, `^`, `&&`, `||`, **`min`**, **`max`**: do the obvious things
  - **`minloc`**, **`maxloc`**: generate a tuple of the min/max and its index
- User-defined:
  - user must define a class that supports a number of methods to:
    - generate a new identity state 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

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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);
```


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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)

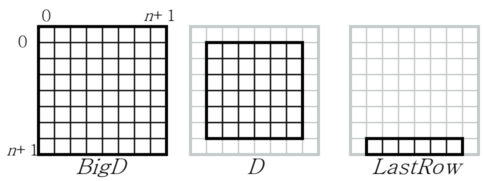
Jacobi Iteration in Chapel

```
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

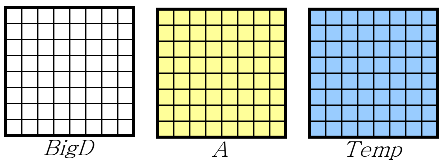
Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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)

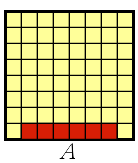


Jacobi Iteration in Chapel

```
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



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

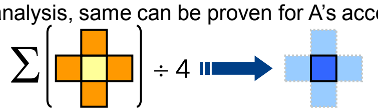
CRAY

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 $(i,j) \in D$ and $D \subseteq BigD$ 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);
```


DARPA HPCS

CRAY

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);
```


DARPA HPCS

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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);
```

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 {
  [(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

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

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

Jacobi Iteration in Chapel

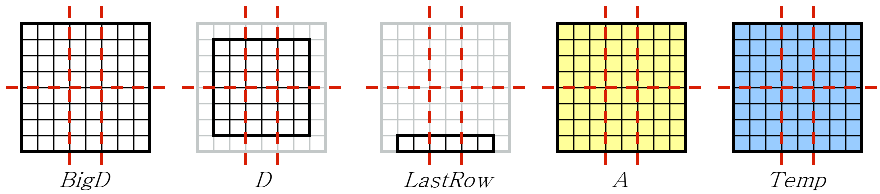
```
config const n = 6,
             epsilon = 1.0e-5;

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

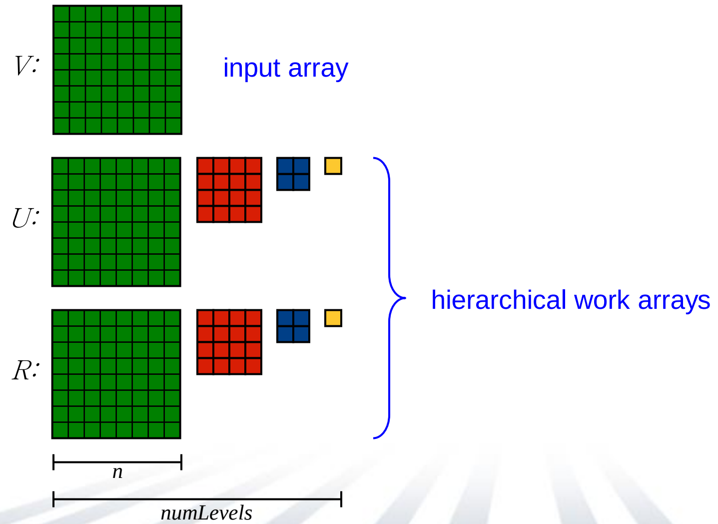


Jacobi Iteration in Chapel

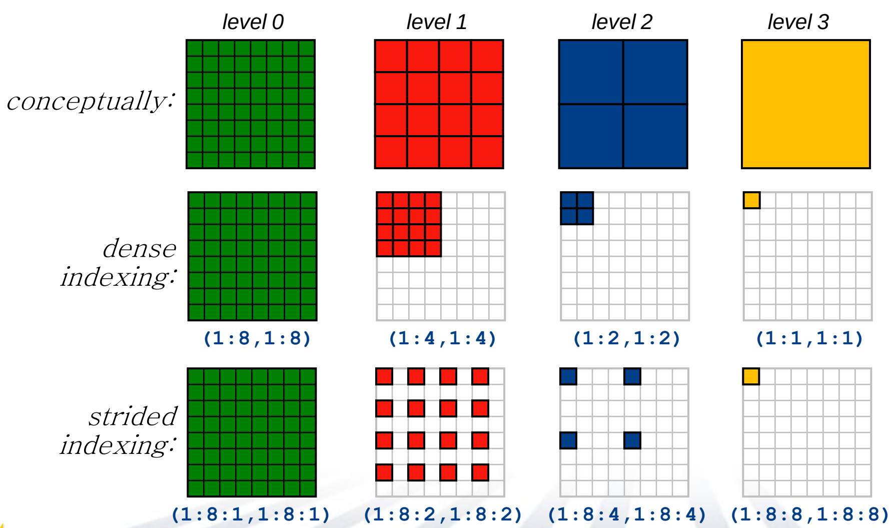
```
config const n = 6,
             epsilon = 1.0e-5;

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


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

Multigrid Example



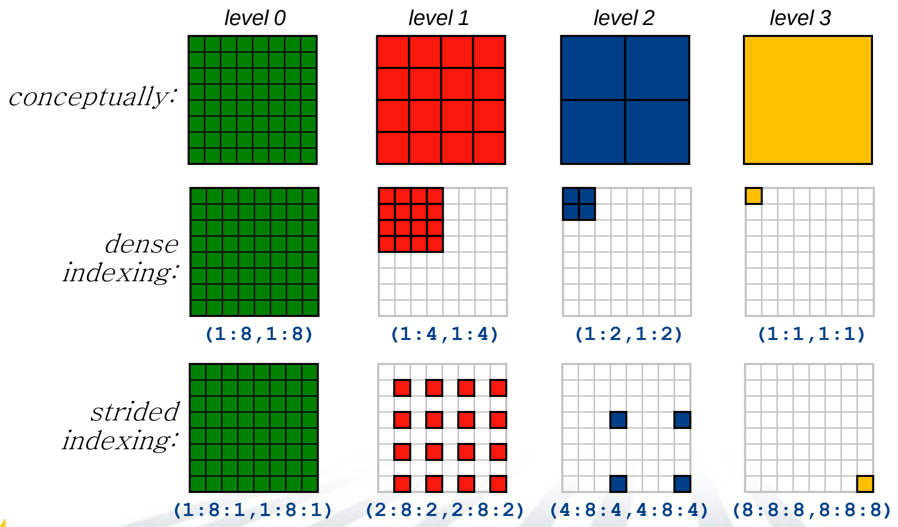
Hierarchical Arrays



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

CRAY

Hierarchical Arrays



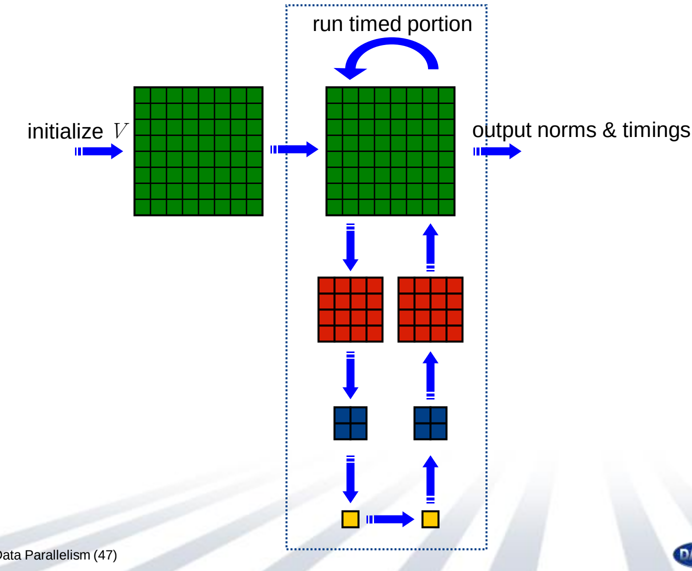
CRAY

Hierarchical Array Declarations in Chapel

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

const Levels = [0..#numLevels];
const ProblemSpace = [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;
```

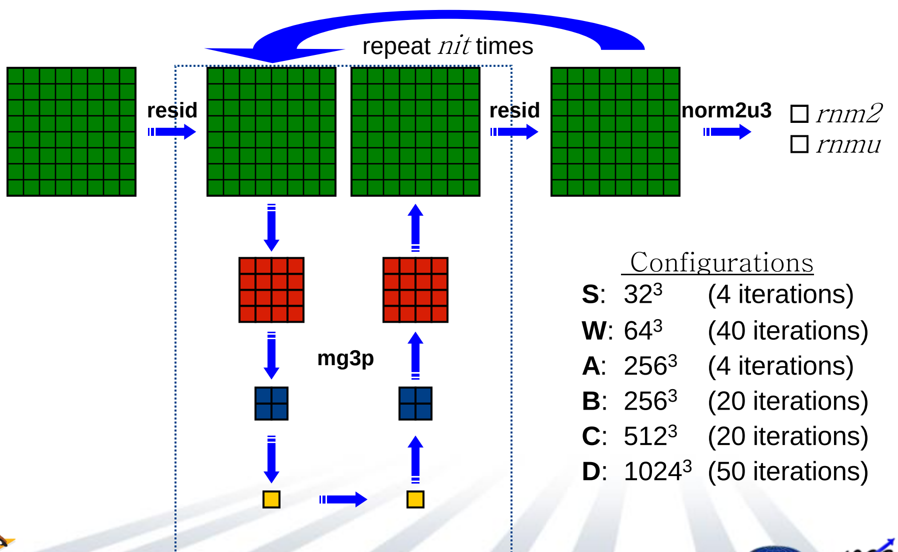

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

Overview of NAS MG



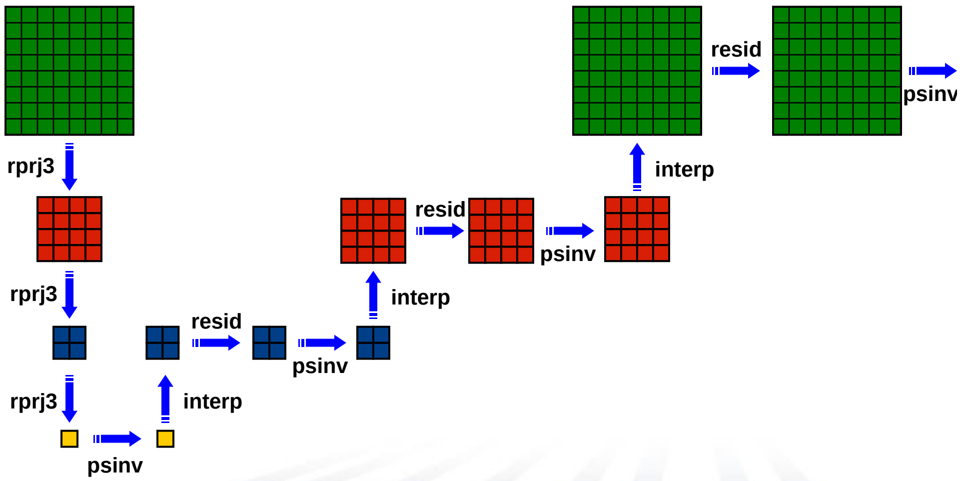
Chapel: Data Parallelism (47)

MG's Timed Portion

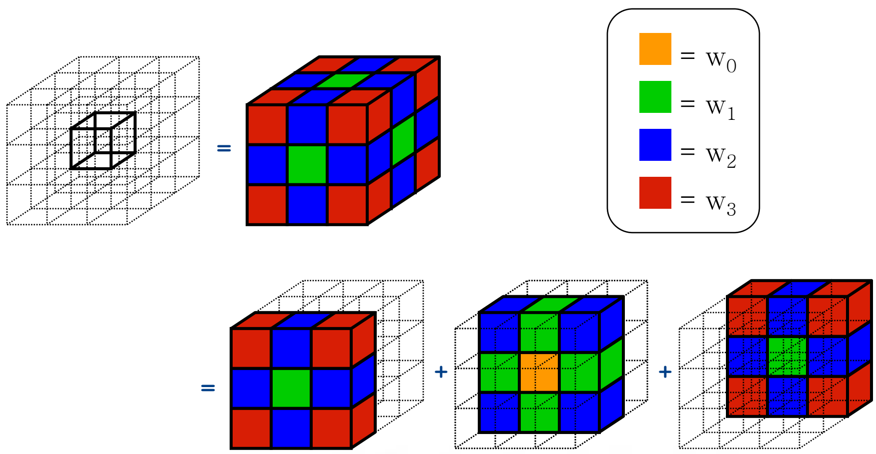


Chapel: Data Parallelism (48)

MG's projection/interpolation cycle



NAS MG: *rprj3* stencil



Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

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], // 27-points
        w: [0..3] real = (0.5, 0.25, 0.125, 0.0625), // 4 weights
        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));
}
```


Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.

NAS MG *rprj3* stencil in Fortran+MPI

Questions?

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.