Data Parallelism

Safe Harbor Statement

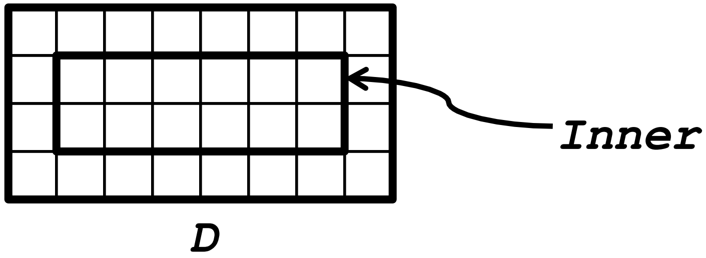
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Domains

Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism

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

const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
```

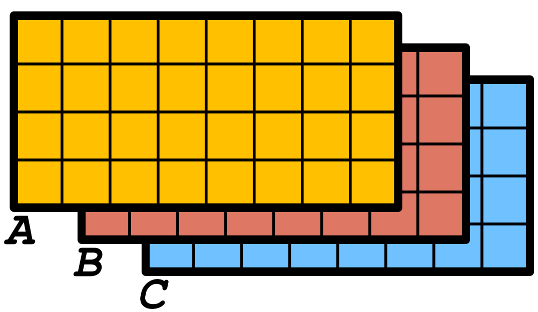


Domains

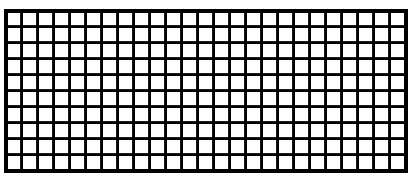
Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism
- Useful for declaring arrays and computing with them

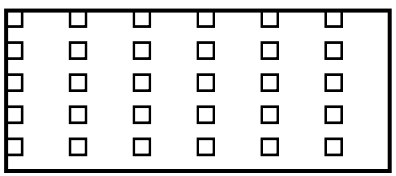
```
config const m = 4, n = 8;

const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};

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



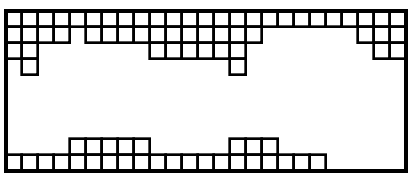
Chapel Domain Types



dense

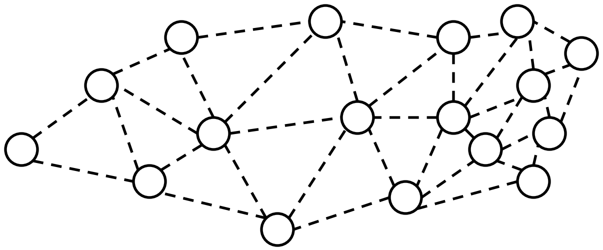


strided



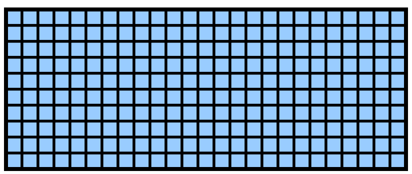
sparse

associative

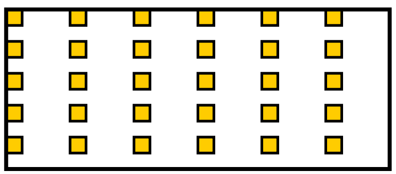


unstructured

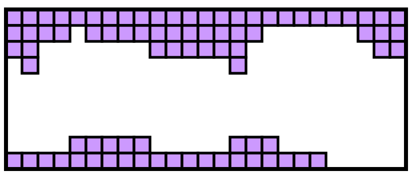
Chapel Array Types



dense

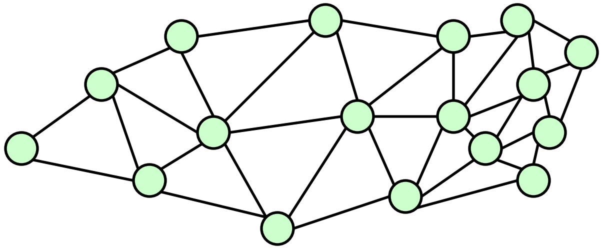


strided



sparse

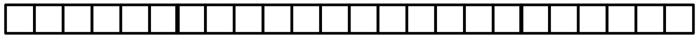
associative



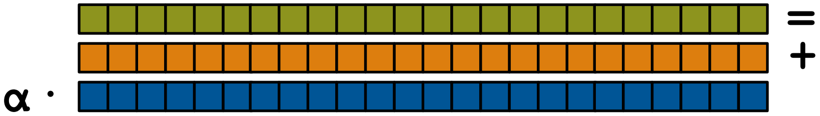
unstructured

Data Parallelism By Example: STREAM Triad

```
const ProblemSpace = {1..m};
```



```
var A, B, C: [ProblemSpace] real;
```




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

Forall Loops

Forall loops: Central concept for data parallel computation

- Like for-loops, but parallel
- Implementation details determined by iterand (e.g., *D* below)
  - e.g., number of tasks, which tasks run which iterations, …
  - in practice, typically uses a number of tasks appropriate for target HW

```
forall (i,j) in D do
  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

- **Forall loops assert…**

…parallel safety: OK to execute iterations simultaneously

…order independence: iterations could occur in any order

…serializability: all iterations could be executed by one task

  - e.g.,, can't have synchronization dependences between iterations

Comparison of Loops: For, Forall, and Coforall

For loops: executed using one task

- use when a loop must be executed serially
- or when one task is sufficient for performance

Forall loops: typically executed using 1 < #tasks << #iters

- use when a loop *should* be executed in parallel…
- …but *can* legally be executed serially
- use when desired # tasks << # of iterations

Coforall loops: executed using a task per iteration

- use when the loop iterations *must* be executed in parallel
- use when you want # tasks == # of iterations
- use when each iteration has substantial work

Forall Intents

- **Tell how to "pass" variables from outer scopes to tasks**
  - Similar to argument intents in syntax and philosophy
    - also adds a "reduce intent", similar to OpenMP
  - Design principles:
    - "principle of least surprise"
    - avoid simple race conditions
    - avoid copies of (potentially) expensive data structures

Forall Intent Examples: Scalars

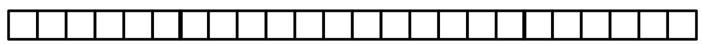
```
var sum: real;
forall i in 1..n do                    // default intent of scalars is 'const in'
  sum += computeMyResult(i);           // so this is illegal (and avoids a race)

var sum: real;
forall i in 1..n with (ref sum) do          // override default intent
  sum += computeMyResult(i);                // we've now requested a race

var sum: real;
forall i in 1..n with (+ reduce sum) do   // override default intent
  sum += computeMyResult(i);   // each task accumulates into its own copy
// on loop exit, all tasks combine their results into original 'sum'
```

Forall Intent Examples: Arrays

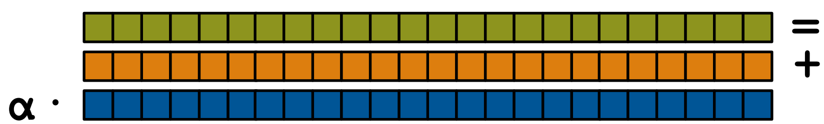
```
var sum: [1..1000] real;
forall i in 1..1000 do                        // default intent for arrays is 'ref'
  sum[i] = computeMyResult(i);                // (avoids array copies by default)

var sum: [1..1000] real;
forall i in 1..1000 with (in sum) do          // override default intent: "copy in"
  sum[i] = computeMyResult(i);                // each task has its own copy now

var sum: [1..1000] real;
forall i in 1..n with (+ reduce sum) do       // request reduce on exit
  sum[computeBucket(i)] += 1;                 // each task has its own copy now
// on loop exit, tasks combine their results into original 'sum', computing a histogram
```

Data Parallelism By Example: STREAM Triad

```
const ProblemSpace = {1..m};
```



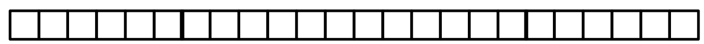
```
var A, B, C: [ProblemSpace] real;
```



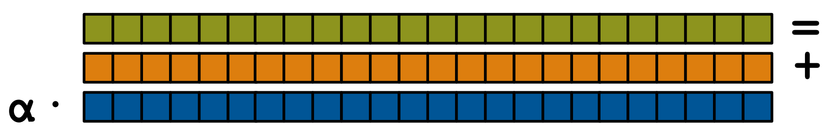
```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha*c;
```

Data Parallelism By Example: STREAM Triad

```
const ProblemSpace = {1..m};
```



```
var A, B, C: [ProblemSpace] real;
```




```
A = B + alpha * C;    // equivalent to the previous zippered forall version
```

Function promotion

- **Scalar functions may be called with array arguments**
  - functions expecting arguments of type *t* can be passed array-of-*t*
    - results in data parallel invocation of function

```
proc foo(x: int, y: int) {
  return 2*x + y;
}
writeln(foo(3,4));                        // prints 10
writeln(foo([1, 2, 4], [2, 3, 4]));   // prints 4  7  12
```

  - Promotion is equivalent to zippered iteration:

```
foo(A, B);
```

==

```
forall (a,b) in zip(A, B) do
  foo(a, b);
```

- **Ranges/domains can also promote functions:**

```
writeln(foo(1..3, 1..6 by 2));   // prints 3 7 11
```

Implication of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

```
A = B + alpha * C;
```

⇒

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

…rather than operator-wise.

```
A = B + alpha * C;
```

~~⇒~~

```
T1 = alpha * C;
A = B + T1;
```

Implication of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

```
A = B + alpha * C;
```

⇒

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

⇒ No temporary arrays required by semantics

- ⇒ No surprises in memory requirements
- ⇒ Friendlier to cache utilization

⇒ Differs from traditional array language semantics

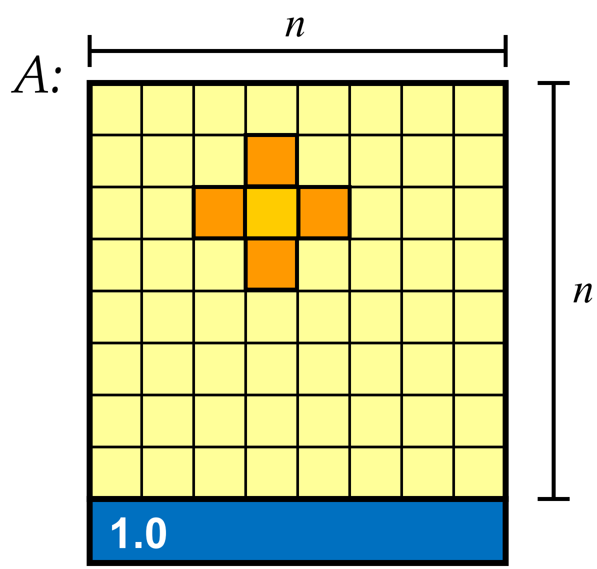
```
A[D] = A[D-one] + A[D+one];
```

Read/write race!

⇒

```
forall (a1, a2, a3)
  in (A[D], A[D-one], A[D+one]) do
    a1 = a2 + a3;
```

Data Parallelism by Example: Jacobi Iteration



repeat until max change < ε

Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1},
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    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);
```

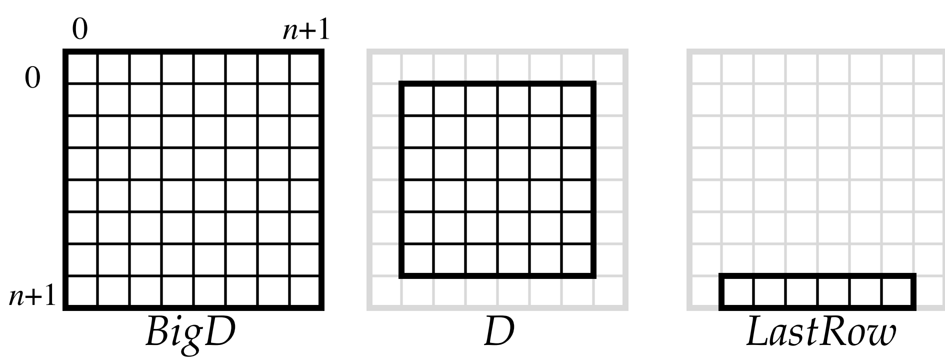
Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

va

A[

do
                                                 j+1]) / 4;

}

wr
```

Declare domains (first class index sets)

{lo..hi, lo2..hi2} ⇒ 2D rectangular domain, with 2-tuple indices

Dom1[Dom2] ⇒ computes the intersection of two domains



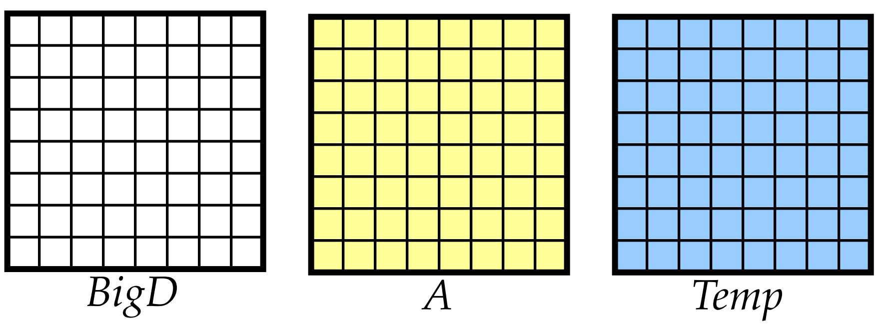
.exterior() ⇒ one of several built-in domain generators

Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1},
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

Declare arrays

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



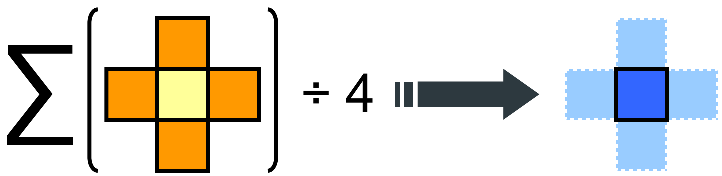

```
[i,j+1]) / 4;
```

Jacobi Iteration in Chapel

```
config const n = 6,
```

Compute 5-point stencil

forall *ind* in *Dom* ⇒ parallel forall expression over *Dom*'s indices, binding them to *ind*
(here, since *Dom* is 2D, we can de-tuple the indices)



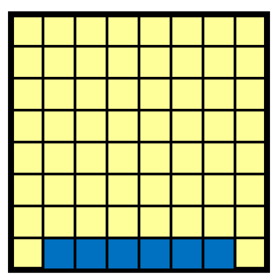

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  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 = {0..n+1, 0..n+1},
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do
                                                      j+1]) / 4;

}

wr
```

Set Explicit Boundary Condition

Arr[Dom] ⇒ refer to array slice (“forall i in Dom do …Arr[i]…”)

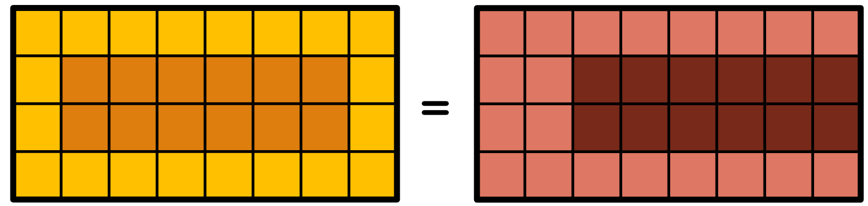


A

Array Slicing

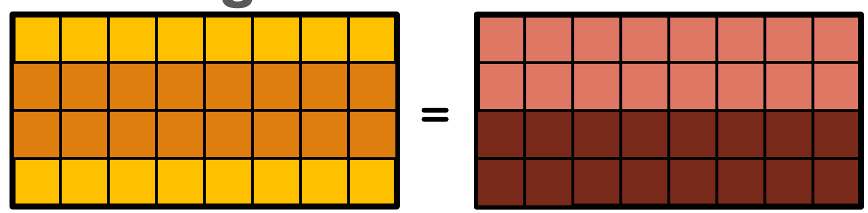
- **Domains can be used to index into arrays**
  - Can be thought of as “promoted array indexing”

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



- **Slices can also be expressed with ranges:**

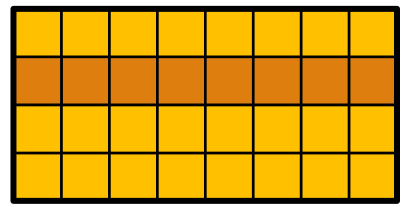
```
A[2..3, ..] = B[3.., 1..n];
```



Rank Change Slicing

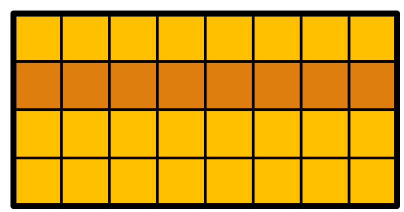
- **Slicing using a 1-element range preserves dimensionality**
  - This is a 2D array expression that's 1 x n:

```
…A[2..2, ..]…
```



- **Slicing using a scalar results in a rank change:**
  - This is a 1D array expression of n-elements:

```
…A[2, ..]…
```



Jacobi Iteration in Chapel

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

Compute maximum change

***op* reduce** ⇒ collapse aggregate expression to scalar using *op*

Promotion: *abs()* and – are scalar operators; providing array operands results in parallel evaluation equivalent to:

```
forall (a,t) in zip(A,Temp) do abs(a – t)
```

```
do {
  forall (i,j) in D do
    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);
```

Reductions in Chapel

- **Standard reductions supported by default:**

```
+, *, min, max, &, |, &&, ||, minloc, maxloc, …
```

- **Reductions can reduce arbitrary iterable expressions:**

```
const total = + reduce Arr,
      factN = * reduce 1..n,
      biggest = max reduce (for i in myIter() do foo(i));
```

- **Advanced users can write their own reductions**
  - However, note that the interface still evolves across releases

Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard *do…while* loop construct

```
do {
  forall (i,j) in D do
    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 = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    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

Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1},
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    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);

use BlockDist;
```

Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1},
           D = BigD[1..n, 1..n],
     LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

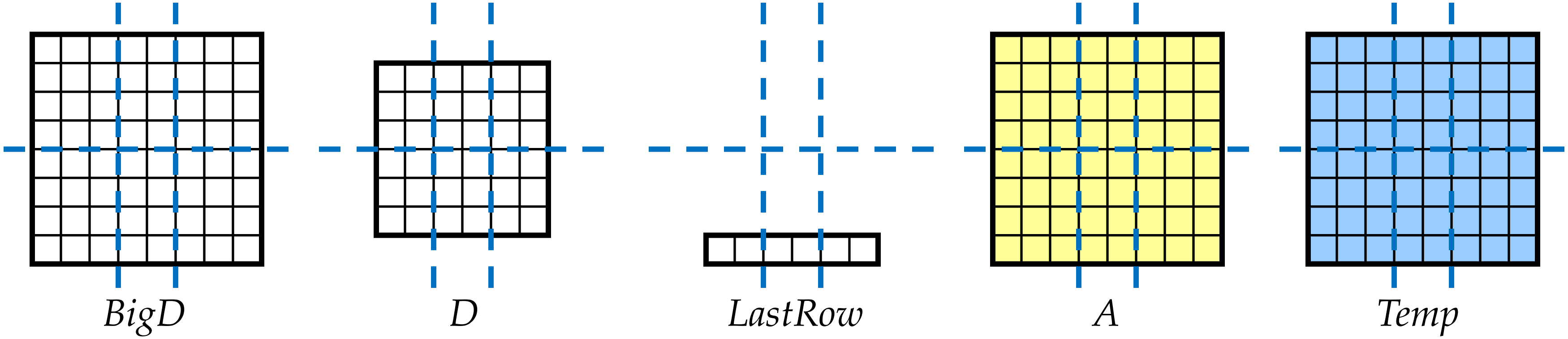
By default, domains and their arrays are mapped to a single locale.
Any data parallelism over such domains/ arrays will be executed by the cores on that locale.
Thus, this is a shared-memory parallel program.

```
    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 = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
         D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this simple change, we specify a mapping from the domains and arrays to locales
Domain maps describe the mapping of domain indices and array elements to *locales*
- specifies how array data is distributed across locales
- specifies how iterations over domains/arrays are mapped to locales



Questions about Data Parallelism?

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.