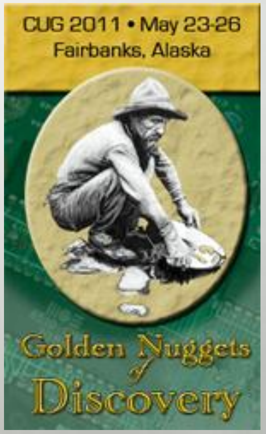
Authoring User-Defined Domain Maps in Chapel

Brad Chamberlain, Sung-Eun Choi, Steve Deitz, David Iten, Vassily Litvinov

Cray Inc.

CUG 2011: May 24th, 2011



What is Chapel?

- A new parallel programming language
  - Design and development led by Cray Inc.
  - Started under the DARPA HPCS program
- **Overall goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Support better portability than current programming models
  - Improve the robustness of parallel codes
- A work-in-progress

Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software
- Target Architectures:
  - multicore desktops and laptops
  - commodity clusters
  - Cray architectures
  - systems from other vendors
  - (in-progress: CPU+accelerator hybrids)

Chapel's High-Level Themes

General Parallel Programming

- "any parallel algorithm on any parallel hardware"

Multiresolution Parallel Programming

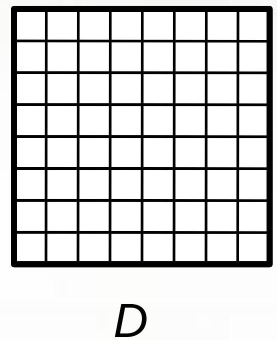
- high-level features for convenience/simplicity
- low-level features for greater control

Control over Locality/Affinity of Data and Tasks

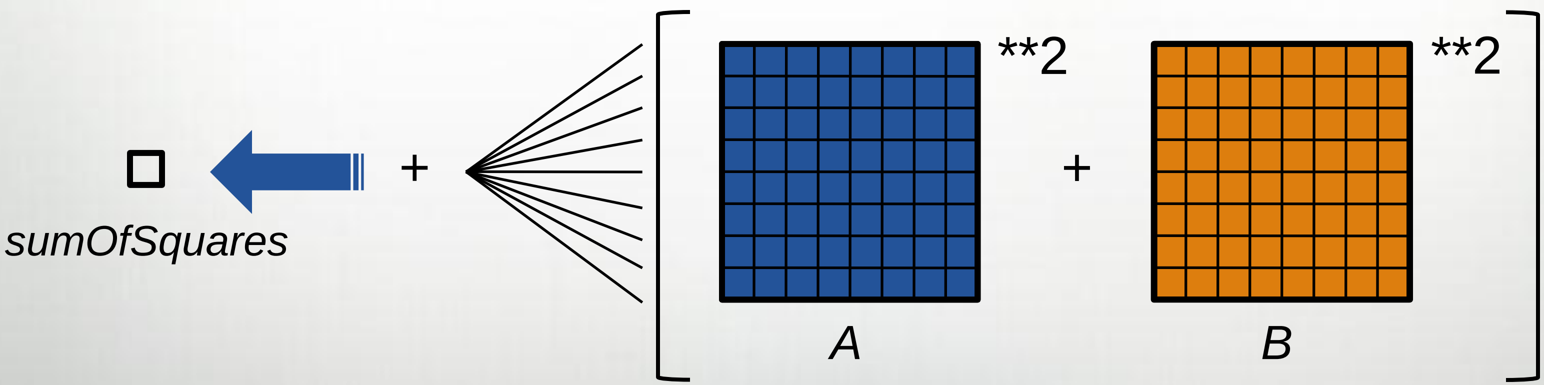
- for scalability

Sample Computation: Sum-of-Squares

```
config const n = computeProblemSize();
const D = [1..n, 1..n];
```



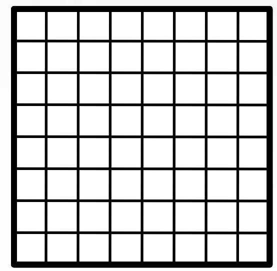

```
var A, B: [D] real;
```




```
const sumOfSquares = + reduce (A**2 + B**2);
```

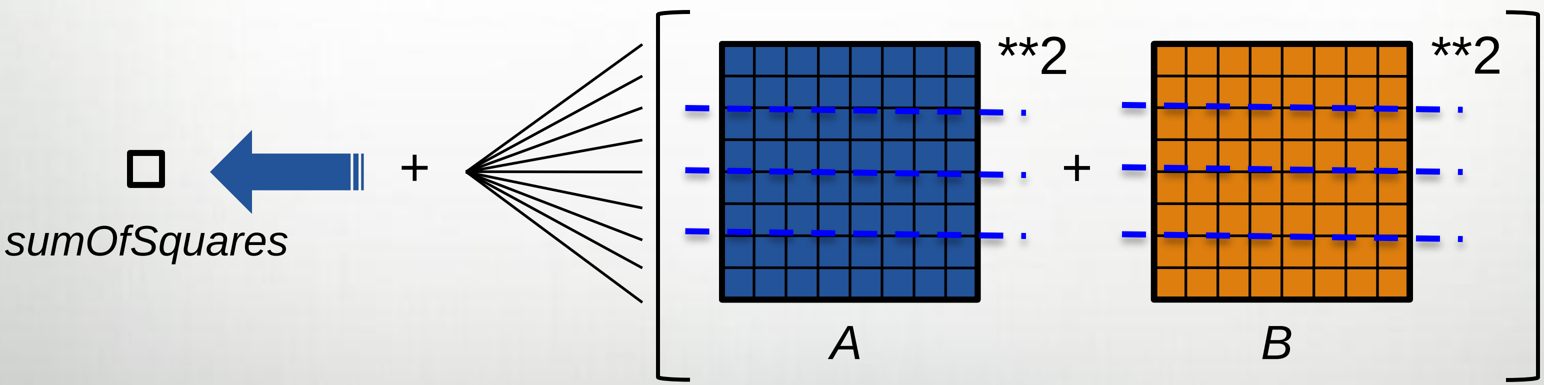
Sample Computation: Sum-of-Squares

```
config const n = computeProblemSize();
const D = [1..n, 1..n];
```



D

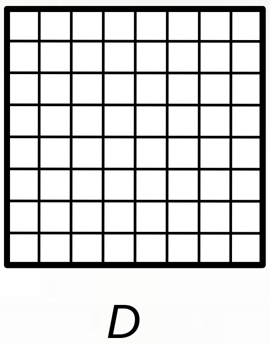
```
var A, B: [D] real;
```



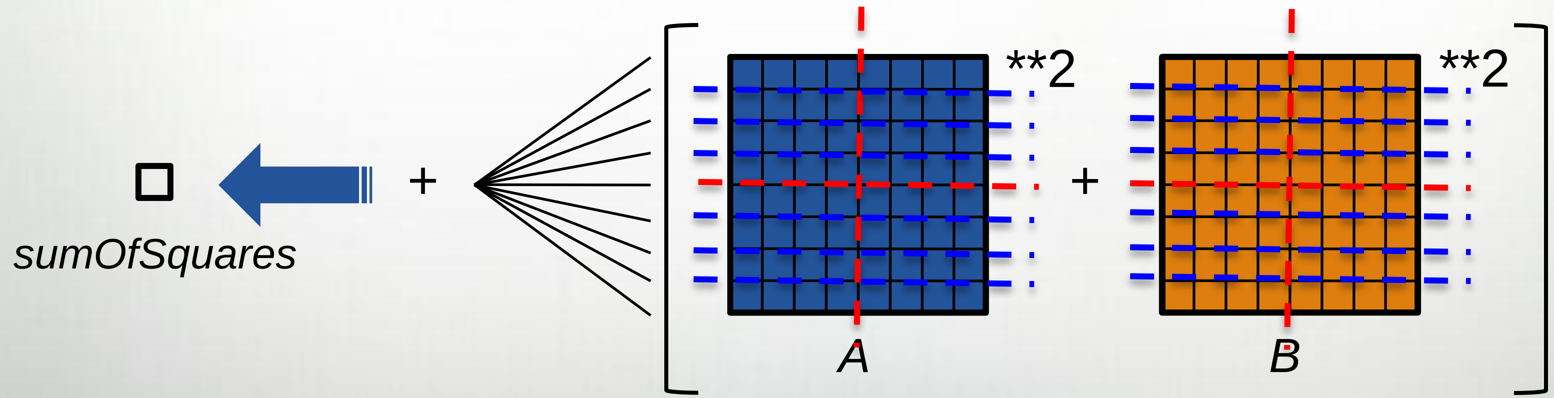

```
const sumOfSquares = + reduce (A**2 + B**2);
```

Sample Computation: Sum-of-Squares

```
config const n = computeProblemSize();
const D = [1..n, 1..n] dmapped …;
```




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




```
const sumOfSquares = + reduce (A**2 + B**2);
```

Sum-of-Squares Implementation

```
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
```

How is this global-view computation implemented in practice?

ZPL: Block-distributed arrays, serial on-node computation (inflexible)

HPF: Not particularly well-defined ("trust the compiler")

Chapel: Very flexible and well-defined via *domain maps* (stay tuned)

Outline

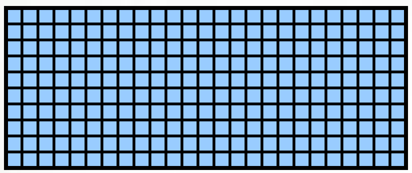
- ✓ Background and Motivation
- Chapel Background:
  - Locales
  - Domains, Arrays, and Domain Maps
- Implementing Domain Maps
- Wrap-up

The Locale Type

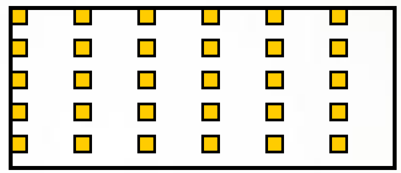
- **Definition**
  - Abstract unit of target architecture
  - Supports reasoning about locality
  - Capable of running tasks and storing variables
    - i.e., has processors and memory
- **Properties**
  - a locale’s tasks have ~uniform access to local vars
  - Other locale’s vars are accessible, but at a price
- **Locale Examples**
  - A multi-core processor
  - An SMP node

Chapel Domain/Array Types

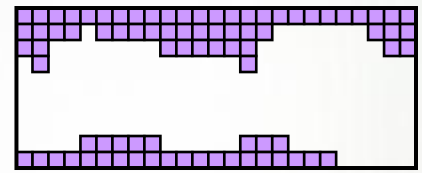
Chapel supports several types of domains and arrays:



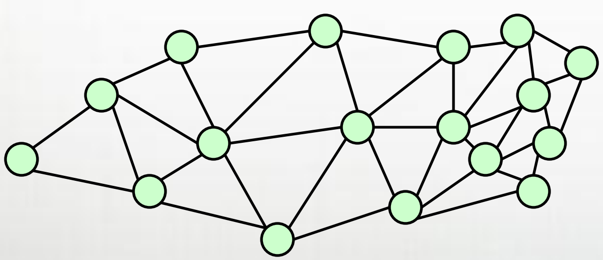
dense



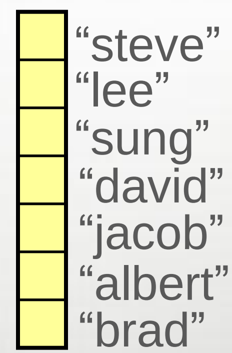
strided



sparse



unstructured



associative

Chapel Domain/Array Operations

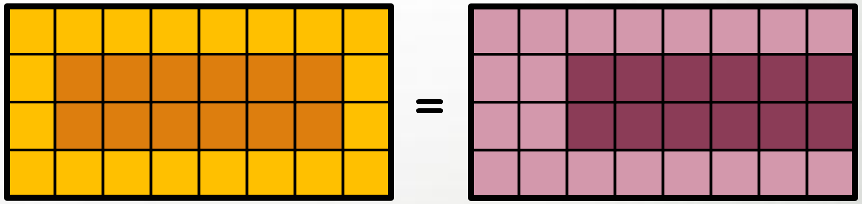
- Whole-Array Operations; Parallel and Serial Iteration

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

- Array Slicing; Domain Algebra

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

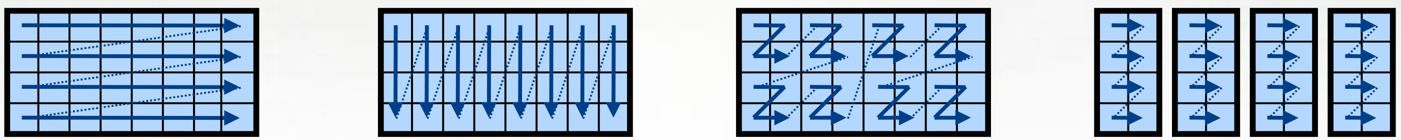


- And several other operations: indexing, reallocation, domain set operations, scalar function promotion, …

Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?

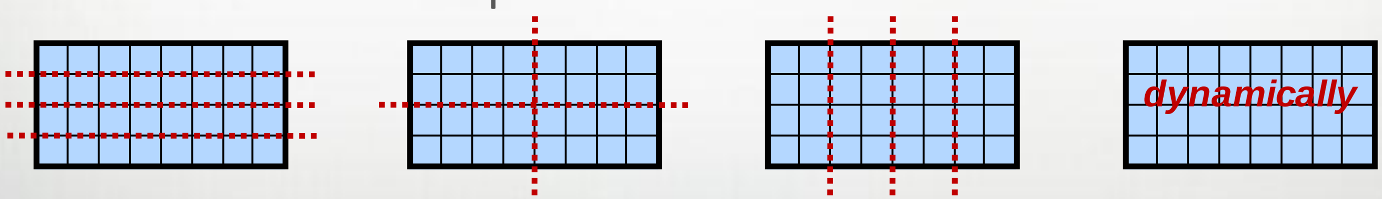


…?

- What data structure is used to store sparse arrays? (COO, CSR, …?)

Q2: How are data parallel operators implemented?

- How many tasks?
- How is the iteration space divided between the tasks?



…?

Data Parallelism: Implementation Qs

Q3: How are arrays distributed between locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

Q4: What architectural features will be used?

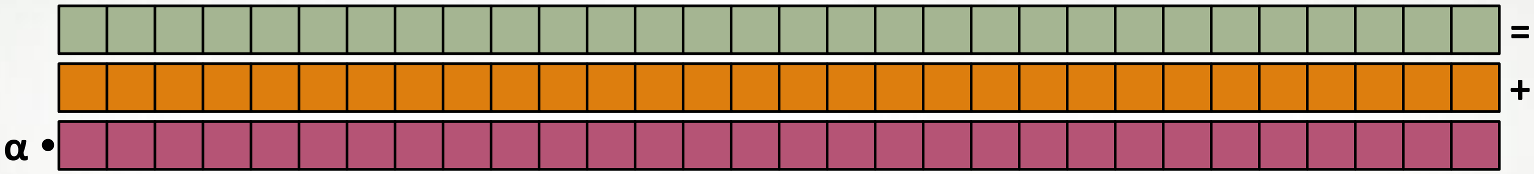
- Can/Will the computation be executed using CPUs? GPUs? both?
- What memory type(s) is the array stored in? CPU? GPU? texture? …?

A1: In Chapel, any of these could be the correct answer

A2: Chapel's *domain maps* are designed to give the user full control over such decisions

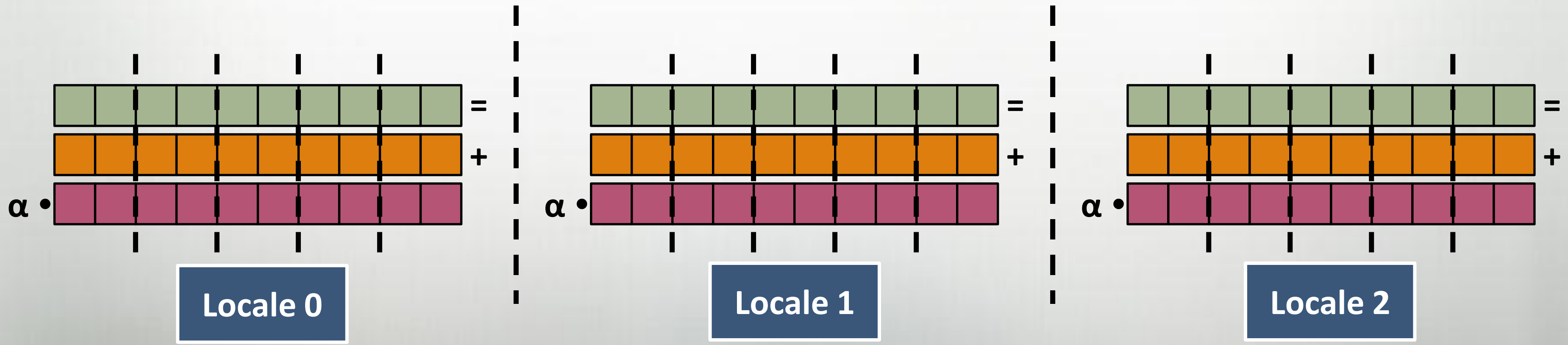
Domain Maps

Domain maps are "recipes" that instruct the compiler how to map the global view of a computation…




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

…to the target locales' memory and processors:



Domain Maps

Domain Maps: "recipes for implementing parallel/ distributed arrays and domains"

They define data storage:

- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale's memory

…as well as operations:

- random access, iteration, slicing, reindexing, rank change, …
- the Chapel compiler generates calls to these methods to implement the user's array operations

Domain Maps: Layouts and Distributions

Domain Maps fall into two major categories:

layouts: target a single locale

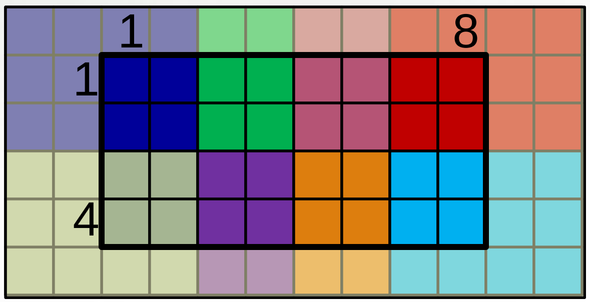
- (that is, a desktop machine or multicore node)
- **examples:** row- and column-major order, tilings, compressed sparse row

distributions: target distinct locales

- (that is a distributed memory cluster or supercomputer)
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, …

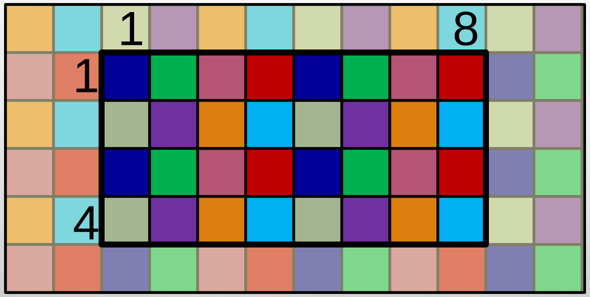
Sample Distributions: Block and Cyclic

```
var Dom = [1..4, 1..8] dmapped Block( [1..4, 1..8] );
```



distributed to

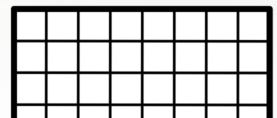

```
var Dom = [1..4, 1..8] dmapped Cyclic( startIdx=(1,1) );
```



distributed to

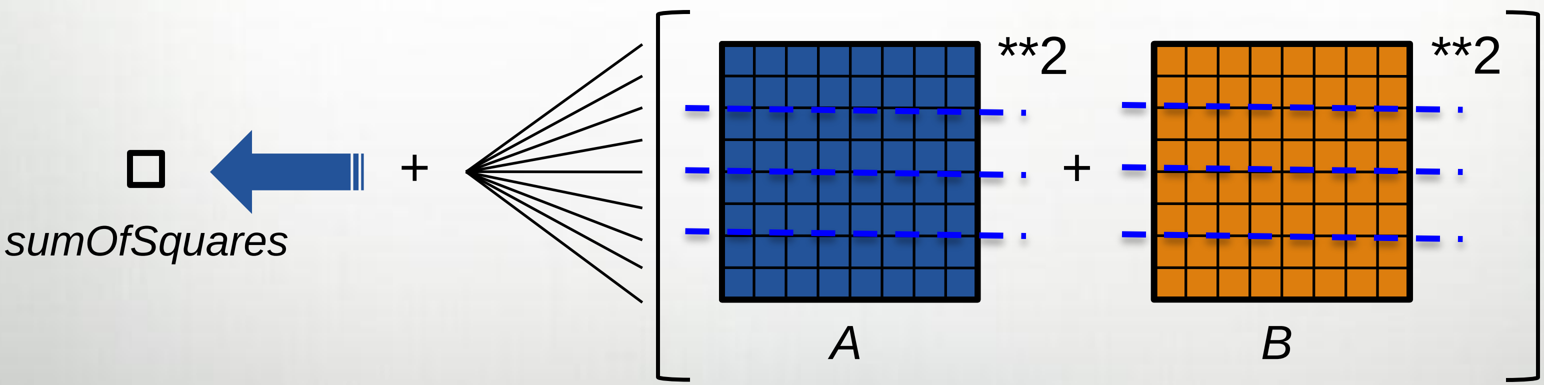
Sample Computation: Local Sum-of-Squares

```
config const n = computeProblemSize();
const D = [1..n, 1..n];
```



```
var A, B: [D] r
```

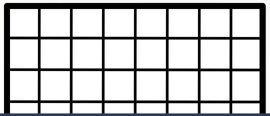
No domain map specified => use default layout
- current locale owns all indices and values
- computation will execute using local resources only




```
const sumOfSquares = + reduce (A**2 + B**2);
```

Sample Computation: Distributed Sum-of-Squares

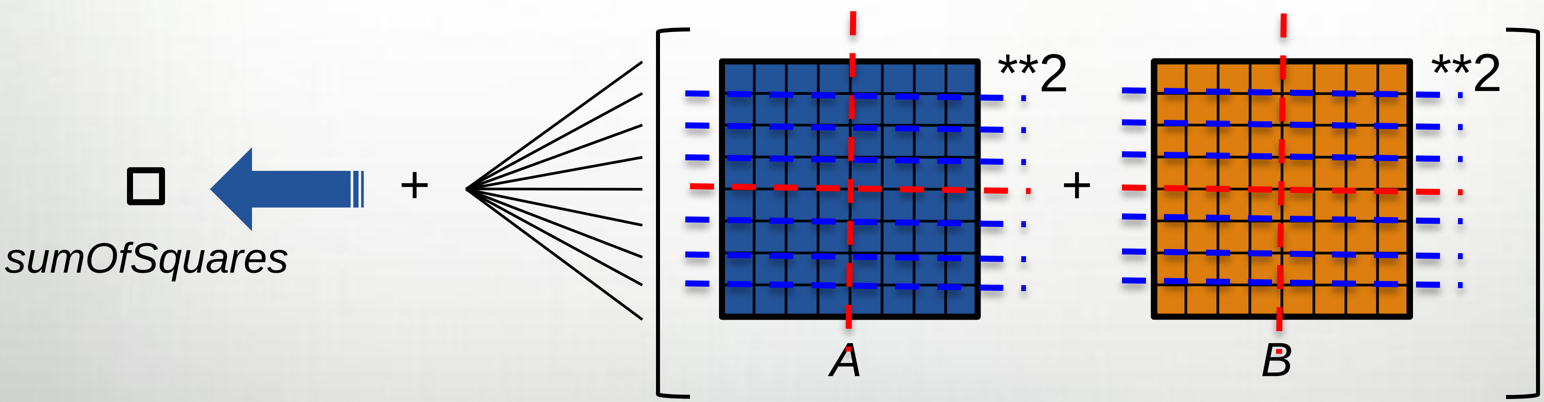
```
config const n = computeProblemSize();
const D = [1..n, 1..n] dmapped Block([1..n, 1..n]);
```



The dmapped keyword specifies a domain map

- “Block” specifies a multidimensional locale blocking
- Each locale stores its local block using the default layout

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



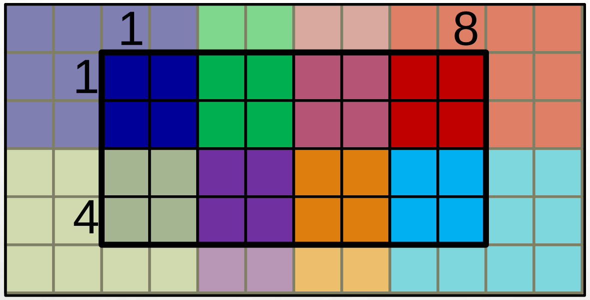

```
const sumOfSquares = + reduce (A**2 + B**2);
```

The Complete Block class constructor

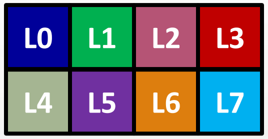
```
proc Block(boundingBox: domain,

           targetLocales: [] locale = Locales,

           dataParTasksPerLocale = ...,
           dataParIgnoreRunningTasks = ...,
           dataParMinGranularity = …)
```

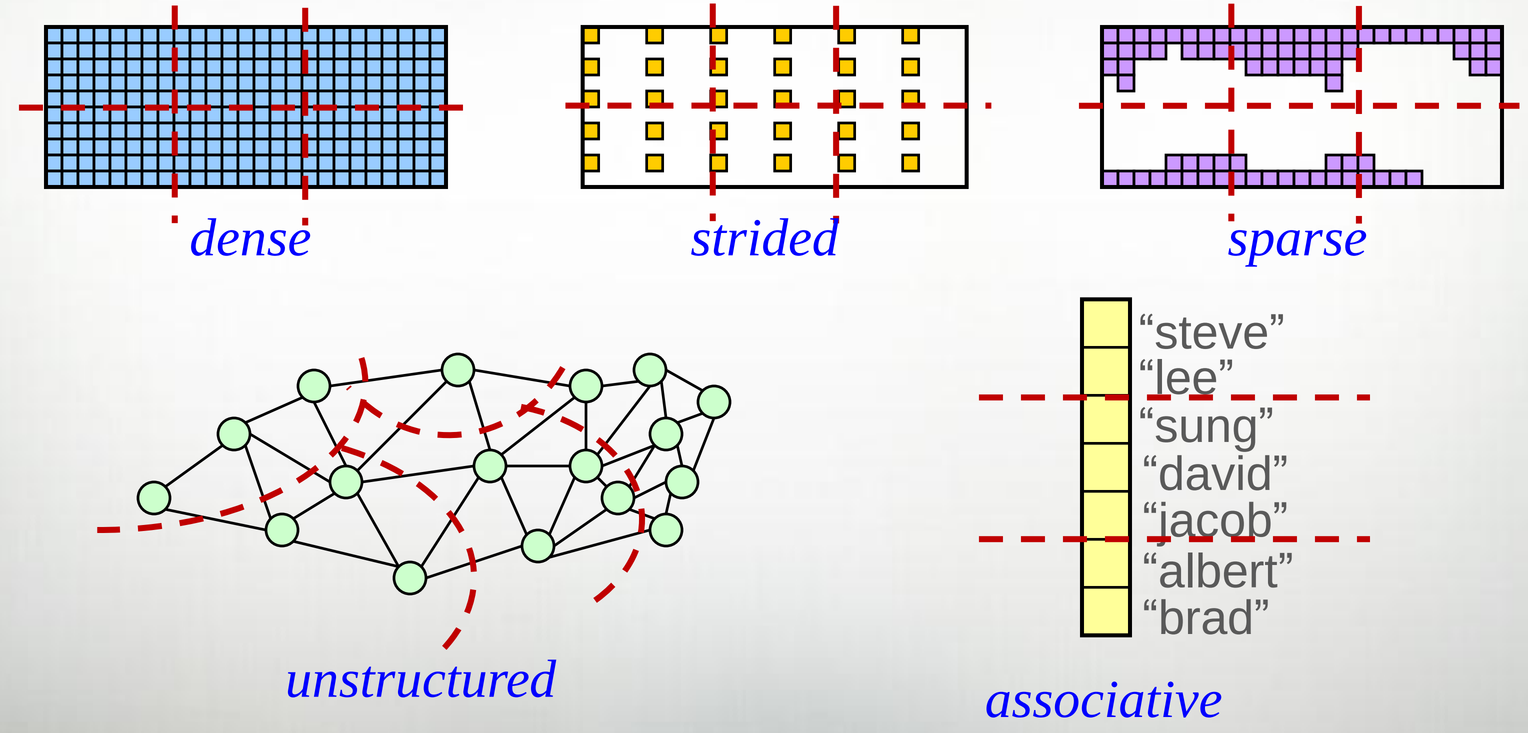


distributed to



All Domain Types Support Domain Maps

All Chapel domain types support domain maps



Outline

- ✓ Background and Motivation
- ✓ Domains, Arrays, and Domain Maps
- Implementing Domain Maps
  - Philosophy
  - Implementing Layouts
  - Implementing Distributions
- Wrap-up

Chapel's Domain Map Philosophy

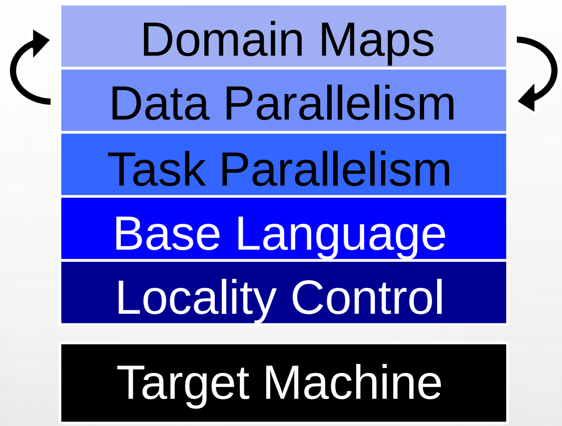
1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly
2. Advanced users can write their own domain maps in Chapel
   - to cope with shortcomings in our standard library
3. Chapel's standard layouts and distributions will be written using the same user-defined domain map framework
   - to avoid a performance cliff between "built-in" and user-defined domain maps
4. Domain maps should only affect implementation and performance, not semantics
   - to support switching between domain maps effortlessly

Chapel's Multiresolution Design

Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control
- build the higher-level concepts in terms of the lower

Chapel language concepts



- separate concerns appropriately for clean design
  - yet permit the user to intermix the layers arbitrarily

Descriptors for Layouts

Domain Map

Represents: a domain map value

Generic w.r.t.: index type

State: the domain map's representation

Typical Size: Θ(1)

Domain

Represents: a domain

Generic w.r.t.: index type

State: representation of index set

Typical Size: Θ(1) → Θ(*numIndices*)

Array

Represents: an array

Generic w.r.t.: index type, element type

State: array elements

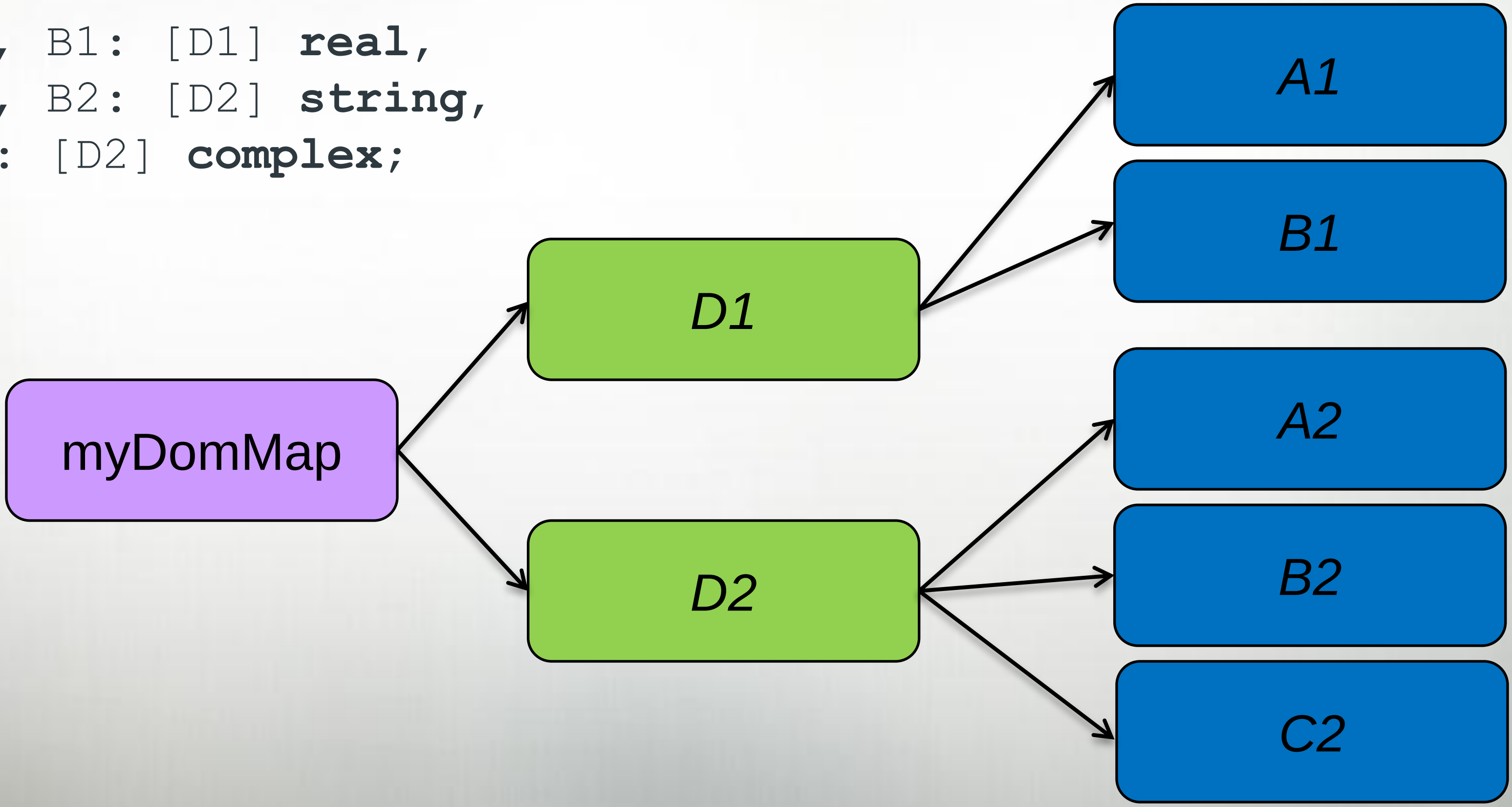
Typical Size: Θ(*numIndices*)

Chapel Declarations and Resulting Descriptors

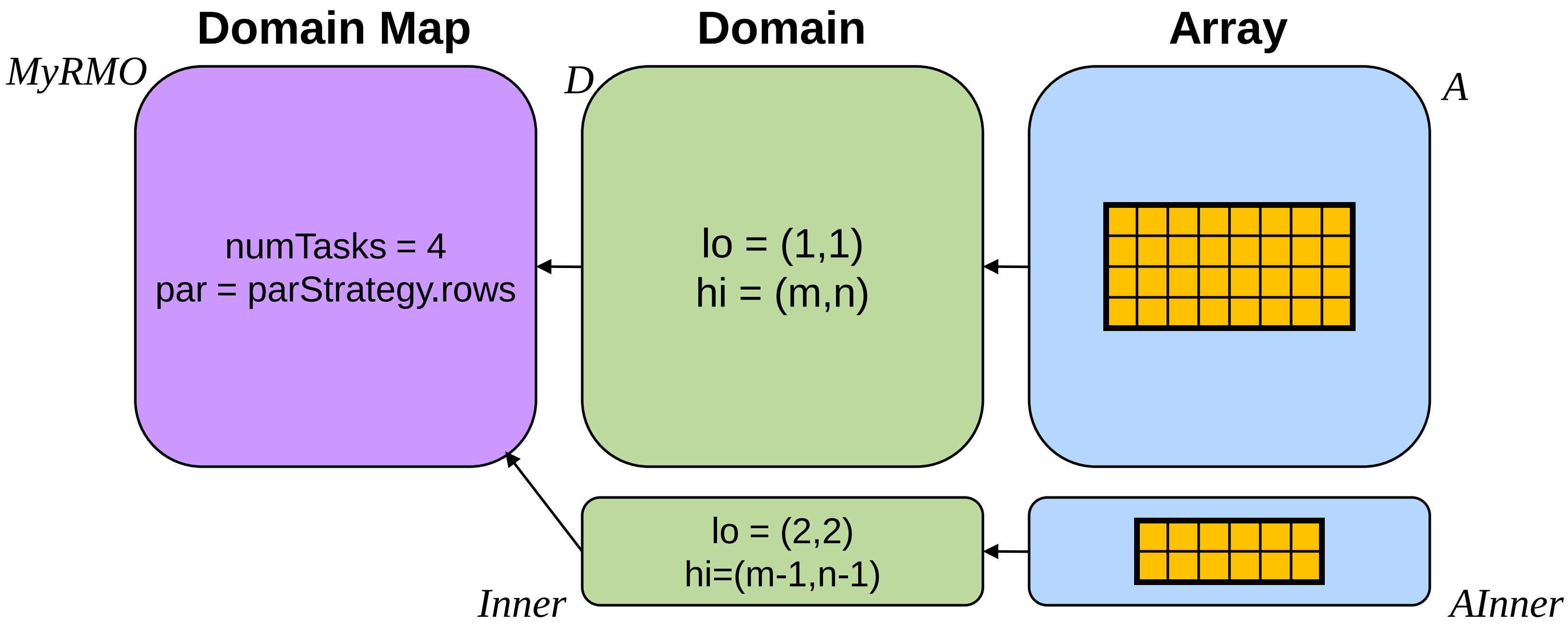
```
const myDomMap = new dmap(DomMapName(args));

const D1 = [1..10] dmapped MyDomMap,
      D2 = [1..20] dmapped MyDomMap;

var A1, B1: [D1] real,
    A2, B2: [D2] string,
    C2: [D2] complex;
```



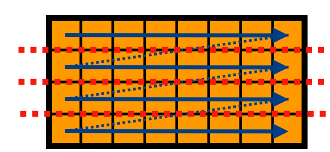
Sample Layout Descriptors



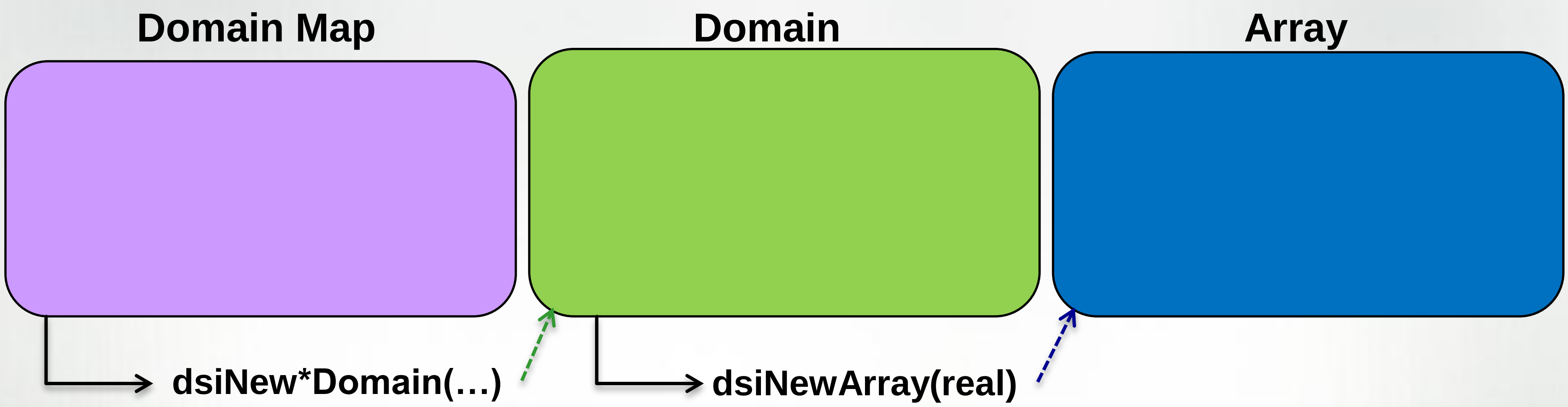

```
const MyRMO = new dmap(new RMO(here.numCores, parStrategy.rows));

const D = [1..m, 1..n] dmapped MyRMO,
      Inner = D[2..m-1, 2..n-1];

var A: [D] real,
    AInner: [Inner] real;
```



Required Descriptor Interfaces: Creation




```
const myDomMap = new dmap(DomMapName(args));
    => myDomMap = new DomMapName(args);

const D1 = [1..10] dmapped MyDomMap;
    => D1 = myDomMap.dsiNewDomain(rank=1, idxType=int);

var A1: [D1] real;
  => A1 = D1.dsiNewArray(real);
```

Required Descriptor Interfaces: Creation

Domain Map | **Domain** | **Array**

dsiIndexToLocale(index): locale

```
…myDomMap.indexToLocale((i,j))…
    => myDomMap.indexToLocale((i,j))
```

Required Descriptor Interfaces: Creation

Domain Map | **Domain** | **Array**

- **dsiNumIndices**(): integer
- **dsiMember**(index): boolean
- *…parallel and serial iterators…*

<u>*regular domains only*</u>

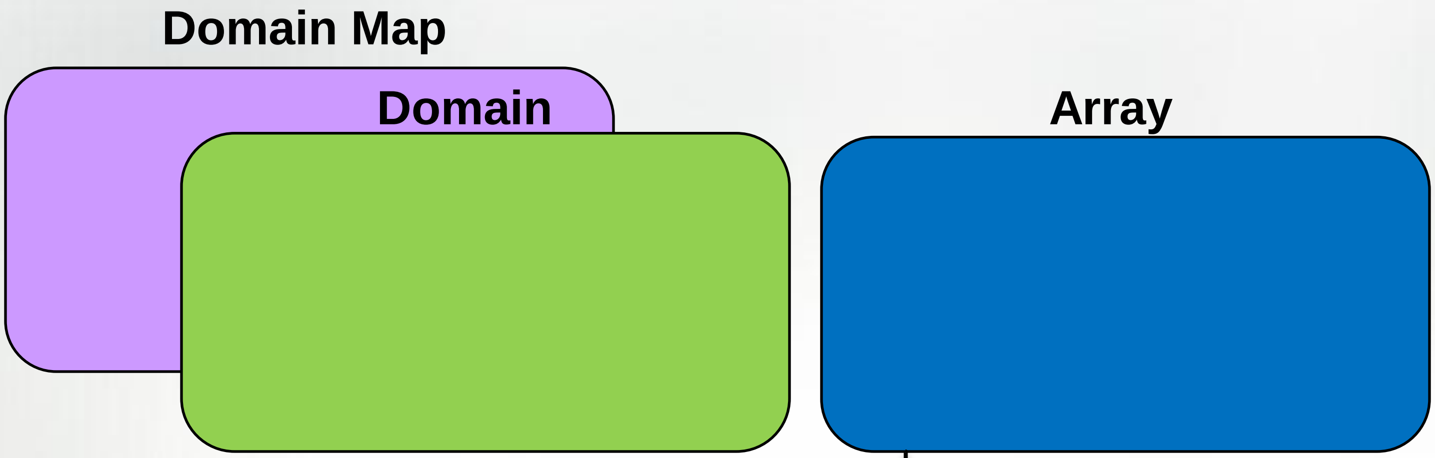
- **dsiGetIndices**(): domain dimensions
- **dsiSetIndices**(domain dimensions)

<u>*irregular domains only*</u>

- **dsiAdd**(index)
- **dsiRemove**(index)
- **dsiClear**()

```
D1 = D2;
  => D1.setIndices(
        D2.getIndices());
```

Required Descriptor Interfaces: Creation



- **dsiAccess**(index): array element
- **dsiSlice**(domain): array descriptor
- **dsiReindex**(domain): array descriptor
- **dsiRankChange**(domain, rank): array descriptor
- *…parallel and serial iterators…*
- …

```
…A1[i,j]…
    => …A1.dsiAccess((i,j))…
```

Distribution Descriptors (One Approach)

	Domain Map	Domain	Array
Global one instance per object (logically)	**Role:** Similar to layout's domain map descriptor **Size:** $\Theta(1) \rightarrow \Theta(\#locales)$	**Role:** Similar to layout's domain descriptor, but no $\Theta(\#indices)$ storage **Size:** $\Theta(1) \rightarrow \Theta(\#locales)$	**Role:** Similar to layout's array descriptor, but data is moved to local descriptors **Size:** $\Theta(1) \rightarrow \Theta(\#locales)$
Local one instance per locale per object (typically)	**Role:** Stores locale-specific domain map parameters **Size:** $\Theta(???)$	**Role:** Stores locale's subset of domain's index set **Size:** $\Theta(1) \rightarrow \Theta(\#indices\ /\ \#locales)$	**Role:** Stores locale's subset of array's elements **Size:** $\Theta(\#indices\ /\ \#locales)$

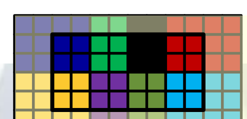
Compiler only knows about global descriptors

so local are just a specific type of state; interface is identical to layouts

Sample Distribution Descriptors

	Domain Map	Domain	Array
Global one instance per object (logically)	boundingBox = [1..4, 1..8] targetLocales = L0 L1 L2 L3 L4 L5 L6 L7	indexSet = [1..4, 1..8]	--
Local one instance per node per object (typically)	L4 myIndexSpace = [3..max, min..2]	L4 myIndices = [3..4, 1..2]	L4 myElems =

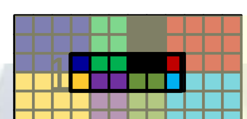
```
var Dom= [1..4, 1..8] dmapped Block(boundingBox=[1..4, 1..8]);
```



Sample Distribution Descriptors

	Domain Map	Domain	Array
Global one instance per object (logically)	boundingBox = [1..4, 1..8] targetLocales = L0 L1 L2 L3 L4 L5 L6 L7	indexSet = [2..3, 2..7]	--
Local one instance per node per object (typically)	L4 myIndexSpace = [3..max, min..2]	L4 myIndices = [3..3, 2..2]	L4 myElems =

```
var Dom= [1..4, 1..8] dmapped Block(boundingBox=[1..4, 1..8]);
var Inner = Dom[2..3, 2..7];
```



Non-Required Descriptor Interfaces

Optional Interfaces

- Do not need to be supplied for correctness
- But supplying them may permit optimizations
- Examples:
  - privatization of global descriptors
  - communication optimizations: stencils, reductions/broadcasts, remaps

User Interfaces

- Add new user methods to domains, arrays
- Not known to the compiler
- Break plug-and-play nature of distributions

Outline

- ✓ Background and Motivation
- ✓ Domains, Arrays, and Domain Maps
- ✓ Implementing Domain Maps
- Wrap-up

Domain Maps: Status

- All Chapel domains and arrays implemented using this framework
  - Full-featured Block, Cyclic, and Replicated distributions
  - COO and CSR Sparse layouts
  - Open addressing quadratic probing Associative layout
  - Block-Cyclic, Dimensional, and Distributed Associative distributions underway
- Initial performance/scaling results promising, but more work remains
- Adding documentation for authoring domain maps

Future Directions

- More advanced uses of domain maps:
  - CPU+GPU cluster programming
  - Dynamic load balancing
  - Resilient computation
  - *in situ* interoperability
  - Out-of-core computations

Summary

- Chapel's domain maps are a promising language concept
  - permit better control over -- and ability to reason about -- parallel array semantics than in previous languages
  - separate specification of an algorithm from its implementation details
  - support a separation of roles:
    - parallel expert writes domain maps
    - parallel-aware computational scientist uses them

For More Information on Domain Maps

- HotPAR'10 paper: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
- This CUG'11 paper
- In the Chapel release…
  - Technical notes detailing the domain map interface for programmers:

    $CHPL_HOME/doc/technotes/README.dsi
  - Browse current domain maps:

    $CHPL_HOME/modules/dists/*.chpl

    layouts/*.chpl

    internal/Default*.chpl

For More Information on Chapel

- **Chapel Home Page** (papers, presentations, tutorials):
  http://chapel.cray.com
- **Chapel Project Page** (releases, source, mailing lists):
  http://sourceforge.net/projects/chapel/
- **General Questions/Info:**
  chapel_info@cray.com (or chapel-users mailing list)

Our Team

- Cray:


Brad Chamberlain


Sung-Eun Choi


Greg Titus


Lee Prokowich


Vass Litvinov


Tom Hildebrandt

- External Collaborators:


Albert Sidelnik


Jonathan Turner


Srinivas Sridharan

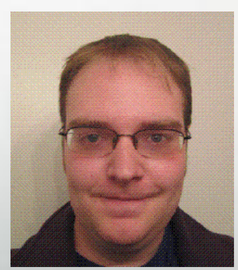
• • •

You? Your Student?

- Interns:


Jonathan Claridge


Hannah Hemmaplardh


Andy Stone


Jim Dinan


Rob Bocchino


Mack Joyner

• • •

THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel-info@cray.com http://sourceforge.net/projects/chapel/