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
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

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

var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
```
config const n = computeProblemSize();
const D = [1..n, 1..n] dmapped ...

var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
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)
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 supports several types of domains and arrays:

- **dense**
- **strided**
- **sparse**
- **unstructured**
- **associative**
Chapel Domain/Array Operations

• Whole-Array Operations; Parallel and Serial Iteration

\[ A = \text{forall} \ (i,j) \text{ in } D \text{ do } (i + j/10.0); \]

• Array Slicing; Domain Algebra

\[ A[\text{InnerD}] = B[\text{InnerD}.\text{translate}(0,1)]; \]

• And several other operations: indexing, reallocation, domain set operations, scalar function promotion, ...
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?

...?
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 are “recipes” that instruct the compiler how to map the global view of a computation...

\[ A = B + \alpha \cdot C; \]

...to the target locales’ memory and processors:
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] );
```

```
var Dom = [1..4, 1..8] dmapped Cyclic( startIdx=(1,1) );
```
Sample Computation: Local Sum-of-Squares

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

var A, B: [D] real;

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

No domain map specified => use default layout
• current locale owns all indices and values
• computation will execute using local resources only
Sample Computation: Distributed Sum-of-Squares

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

var A, B: [D]

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

The `dmapped` keyword specifies a domain map
• “Block” specifies a multidimensional locale blocking
• Each locale stores its local block using the default layout
The Complete Block class constructor

```plaintext
proc Block(boundingBox: domain,

    targetLocales: [] locale = Locales,

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

distributed to: L0, L1, L2, L3, L4, L5, L6, L7
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
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
**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*

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- 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:** $\Theta(1)$

**Domain**
- **Represents:** a domain
- **Generic w.r.t.:** index type
- **State:** representation of index set
- **Typical Size:** $\Theta(1) \rightarrow \Theta(numIndices)$

**Array**
- **Represents:** an array
- **Generic w.r.t.:** index type, element type
- **State:** array elements
- **Typical Size:** $\Theta(numIndices)$
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;
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;
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

dsiIndexToLocale(index): locale

...myDomMap.indexToLocale((i,j))...

=> myDomMap.indexToLocale((i,j))
Required Descriptor Interfaces: Creation

\[
D1 = D2; \\
\rightarrow D1.setIndices(D2.getIndices());
\]

\[\text{regular domains only}\]
- \text{dsiNumIndices()}: integer
- \text{dsiMember(index)}: boolean
- \text{…parallel and serial iterators…}

\[\text{irregular domains only}\]
- \text{dsiGetIndices()}: domain dimensions
- \text{dsiSetIndices(domain dimensions)}
- \text{dsiAdd(index)}
- \text{dsiRemove(index)}
- \text{dsiClear()}
Required Descriptor Interfaces: Creation

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

dsiAccess(index): array element

dsiSlice(domain): array descriptor

dsiReindex(domain): array descriptor

dsiRankChange(domain, rank): array descriptor

...parallel and serial iterators...
Distribution Descriptors (One Approach)

**Global**
- Role: Similar to layout’s domain map descriptor
- Size: $\Theta(1) \rightarrow \Theta(\#\text{locales})$

**Local**
- Role: Stores locale-specific domain map parameters
- Size: $\Theta(???)$

**Domain Map**
- Role: Similar to layout’s domain descriptor, but no $\Theta(\#\text{indices})$ storage
- Size: $\Theta(1) \rightarrow \Theta(\#\text{locales})$

**Domain**
- Role: Stores locale’s subset of domain’s index set
- Size: $\Theta(1) \rightarrow \Theta(\#\text{indices} / \#\text{locales})$

**Array**
- Role: Stores locale’s subset of array’s elements
- Size: $\Theta(\#\text{indices} / \#\text{locales})$

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**
- **Global**
  - one instance per object (logically)
  - boundingBox = \([1..4, 1..8]\)
  - targetLocales = \([L0, L1, L2, L3, L4, L5, L6, L7]\)

**Local**
- one instance per node per object (typically)
  - myIndexSpace = \([3..max, min..2]\)

**Domain**
- indexSet = \([1..4, 1..8]\)

**Array**
- --

```javascript
var Dom= [1..4, 1..8] dmapped Block(boundingBox=[1..4, 1..8]);
```
**Sample Distribution Descriptors**

### Domain Map
- **Global**
  - one instance per object (logically)
  - `boundingBox = [1..4, 1..8]`
  - `targetLocales = [L0, L1, L2, L3, L4, L5, L6, L7]`

### Domain
- **Index Set**
  - `indexSet = [2..3, 2..7]`

### Array
- `myElems = [L4]`

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

---

**Local**
- one instance per node per object (typically)
- `myIndexSpace = [3..max, min..2]`
- `myIndices = [3..3, 2..2]`
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
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](http://chapel.cray.com)
- **General Questions/Info:** [chapel_info@cray.com](mailto: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

- **Interns:**
  - Jonathan Claridge
  - Hannah Hemmaplardh
  - Andy Stone
  - Jim Dinan
  - Rob Bocchino
  - Mack Joyner