Chapel: Domain Maps

(Layouts and Distributions)
"Hello World" in Chapel: a Domain-Map Version

- Multi-locale Data Parallel Hello World

```chapel
config const numIters = 100000;
const WorkSpace = {1..numIters} dmapped Block(...);

forall i in WorkSpace do
  writeln("Hello, world! ",
         "from iteration ", i, " of ", numIters,
         " on locale ", here.id, " of ", numLocales);
```
Review: Data Parallelism

- Domains are first-class index sets
- Specify the size and shape of arrays
- Support iteration, array operations, etc.

![Diagram showing domains and arrays](image)
**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?

* …?

* dynamically
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” (written in Chapel) 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

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
const ProblemSpace = {1..m};

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

A = B + alpha * C;

No domain map specified => use default layout
• current locale owns all indices and values
• computation will execute using local processors only
const ProblemSpace = {1..m}

\texttt{dmapped} Block(boundingBox={1..m});

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

A = B + alpha * C;
const ProblemSpace = \{1..m\}

dmapped Cyclic(startIdx=1);

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

A = B + alpha \cdot C;
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 multiple locales
- (that is a distributed memory cluster or supercomputer)
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, ...
Declaring a Distributed Domain

- Domain types and literals may be domain mapped
  - In practice, this tends to be a great place to rely on type inference to avoid repetition:

```chapel
const Dom = {1..m, 1..n} dmapped myDMap(...);
```

- Domain maps can also be declared independently of a domain value (not covered here)
  - Useful for declaring several domains using the same map
Some Standard Distributions: Block and Cyclic

```plaintext
var Dom = {1..4, 1..8} dmapped Block(boundingBox={1..4, 1..8});
```

```
var Dom = {1..4, 1..8} dmapped Cyclic(startIdx=(1,1));
```

![Block distribution diagram](image1)

```
distributed to
```

![Cyclic distribution diagram](image2)

```
distributed to
```

![Block distribution diagram](image3)

```
distributed to
```

![Cyclic distribution diagram](image4)
The Block class constructor

```plaintext
proc Block(boundingBox: domain,
    targetLocales: [] locale = Locales,
    dataParTasksPerLocale = ...,
    dataParIgnoreRunningTasks = ...,
    dataParMinGranularity = ...)
```
The Cyclic class constructor

```plaintext
proc Cyclic(startIdx,
    targetLocales: [] locale = Locales,
    dataParTasksPerLocale = ...,
    dataParIgnoreRunningTasks = ...,
    dataParMinGranularity = ...)
```

distributed to

```
  L0  L1  L2  L3
  L4  L5  L6  L7
```
All Domain Types Support Domain Maps

- **dense**
- **strided**
- **sparse**
- **unstructured**
- **associative**

- “steve”
- “lee”
- “sung”
- “david”
- “jacob”
- “albert”
- “brad”
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 the standard library

3. Chapel’s standard domain maps are written using the same end-user framework
   • to avoid a performance cliff between “built-in” and user-defined cases
For More Information on Domain Maps


CUG 2011: *Authoring User-Defined Domain Maps in Chapel*, Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:

- Technical notes detailing domain map interface for programmers:
  
  `$CHPL_HOME/doc/technotes/README.dsi`

- Current domain maps:
  
  `$CHPL_HOME/modules/dists/*.*.chpl`
  
  `$CHPL_HOME/modules/layouts/*.*.chpl`
  
  `$CHPL_HOME/internal/Default/*.*.chpl`
Domain Maps: Status

- Full-featured Block, Cyclic, Replicated distributions
- COO and CSR Sparse layouts supported
- Quadratic probing Associative layout supported
- Prototype Block-Cyclic and 2D Dimensional distribution available
- Associative distributions underway
- User-defined domain map interface still evolving
- Memory currently leaked for distributed arrays
Future Directions

• Advanced uses of domain maps:
  • GPU programming
  • Dynamic load balancing
  • Resilient computation
  • *in situ* interoperability
  • Out-of-core computations

• Improved syntax for declared domain maps