Chapel: Multi-Locale Execution

The Locale Type

- Definition
  - Abstract unit of target architecture
  - Capacity for processing and storage (memory)
  - Supports reasoning about locality
- Properties
  - Locale’s tasks have uniform access to local memory
  - Other locale’s memory is accessible, but at a price
- Examples
  - A multi-core processor
  - An XMT

Program Startup

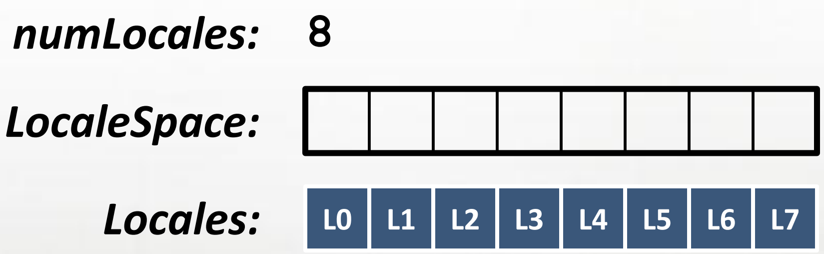
- Execution Context

```
config const numLocales: int;
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

- Specify # of locales when running executable

```
% a.out --numLocales=8
```

```
% a.out -nl 8
```



- Execution begins as a single task on a locale 0

Locale Methods

- ```
  def locale.id: int { ... }
  ```
  Returns index in LocaleSpace
- ```
  def locale.name: string { ... }
  ```
  Returns name of locale (like uname -a)
- ```
  def locale.numCores: int { ... }
  ```
  Returns number of cores available to locale
- ```
  def locale.physicalMemory(...) { ... }
  ```
  Returns physical memory available to user programs on locale

  Example
  ```
  const totalPhysicalMemory =
    + reduce Locales.physicalMemory();
  ```

The On Statement

- Syntax

```
on-stmt:
  on expr { stmt }
```

- Semantics
  - Executes *stmt* on the locale that stores *expr*
  - Does not introduce concurrency
- Example

```
var A: [LocaleSpace] int;
coforall loc in Locales do
  on loc do
    A(loc.id) = compute(loc.id);
```

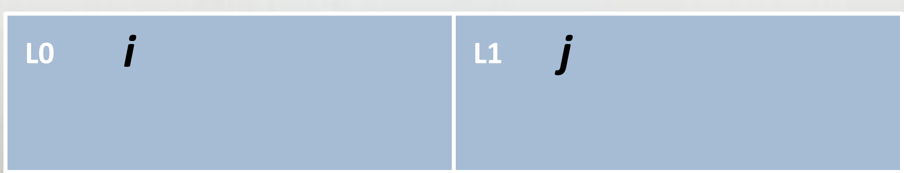
Querying a Variable's Locale

- Syntax

```
locale-query-expr:
  expr . locale
```

- Semantics
  - Returns the locale on which *expr* is stored
- Example

```
var i: int;
on Locales(1) {
  var j: int;
  writeln(i.locale.id, j.locale.id); // outputs 01
}
```



Here

- Built-in locale

```
const here: locale;
```

- Semantics
  - Refers to the locale on which the task is executing
- Example

```
writeln(here.id);      // outputs 0
on Locales(1) do
  writeln(here.id);    // outputs 1
```

Serial Example with Implicit Communication

```
var x, y: real;          // x and y allocated on locale 0

on Locales(1) {          // migrate task to locale 1
  var z: real;           // z allocated on locale 1

  z = x + y;             // remote reads of x and y

  on Locales(0) do       // migrate back to locale 0
    z = x + y;           // remote write to z
                         // migrate back to locale 1
  on x do                // data-driven migration to locale 0
    z = x + y;           // remote write to z
                         // migrate back to locale 1
}                        // migrate back to locale 0
```

L0	L1
x ***y***	***z***

Multi-Locale Examples

- examples/primers/multilocale.chpl

Outline

- Multi-Locale Basics
- Data Parallelism Revisited
- Domain Maps
- Chapel Standard Layouts and Distributions
- User-defined Domain Maps

Flashback: Data Parallelism

- Domain are first class index sets
  - Specifies size and shape of arrays
  - Supports iteration, array operations, etc.
- Arrays are defined using Domains

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?

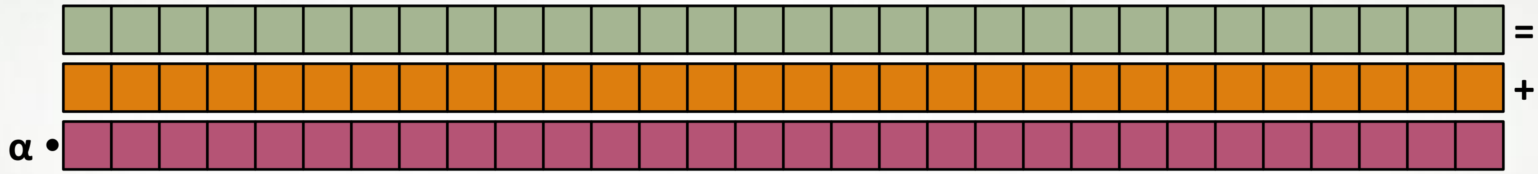
A: Chapel's *domain maps* are designed to give the user full control over such decisions

Outline

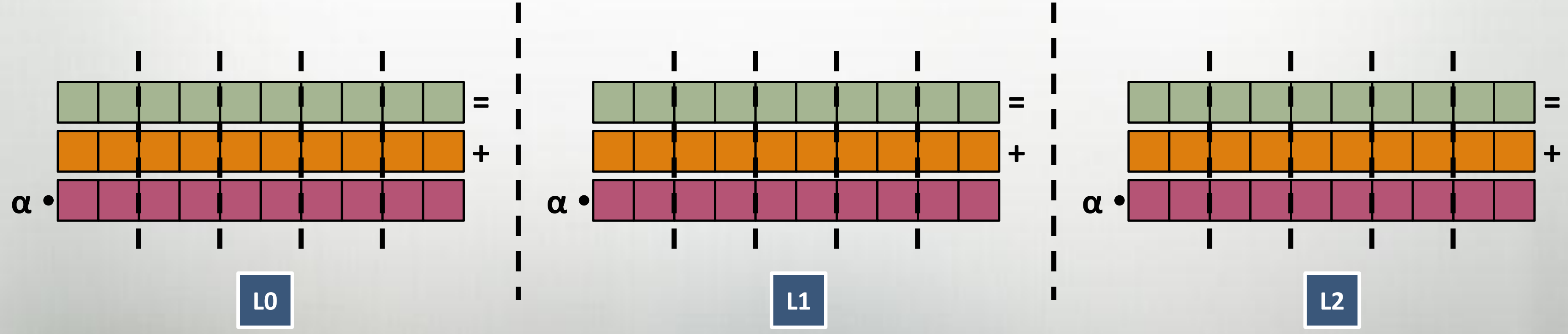
- Multi-Locale Basics
- Data Parallelism Revisited
- Domain Maps
  - Layouts
  - Distributions
- Chapel Standard Layouts and Distributions
- User-defined Domain Maps

Domain Maps

Domain maps are a "recipe" that instructs the compiler how to map the global view…



…to memory and/or locales



More on Domain Maps

A domain map defines:

- Ownership of domain indices and array elements
- Underlying representation
- Standard set of operations on domains and arrays
  - E.g, slicing, reindexing, rank change
- How to farm out work
  - E.g., forall loops over distributed domains/arrays

Domain maps are built using language-level constructs

Domain Maps: Layouts and Distributions

Domain Maps fall into two categories:

layouts: target a single shared memory segment

- *e.g.*, a desktop machine or multicore node

distributions: target multiple distinct memory segments

- *e.g.*, a distributed memory cluster or supercomputer

- Most of our work to date has focused on distributions

Layouts

Layouts are single-locale domain maps

- Uses task parallel constructs to implement data parallelism
- May take advantage of locale resources, *e.g.*, multiple cores

Examples

- Sparse CSR
- GPU

Distributions

Distributions are multi-locale domain maps

- Uses task parallel constructs to implement data parallelism
- Uses on to control data and task locality
- May use layouts for per-locale implementation

Examples

- Block
- Cyclic
- Block-Cyclic
- Block CSR
- Recursive bisection

Chapel's Domain Map Strategy

- Chapel provides a library of standard domain maps
  - to support common array implementations effortlessly
- Advanced users can write their own domain maps in Chapel
  - to cope with shortcomings in our standard library
- 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
- Domain maps should only affect implementation and performance, not semantics
  - to support switching between domain maps effortlessly

Using Domain Maps

- Syntax

```
dmap-type:
  dmap(dmap-class(...))
dmap-value:
  new dmap(new dmap-class(...))
```

- Semantics
  - Domain map classes are defined in Chapel
- Examples

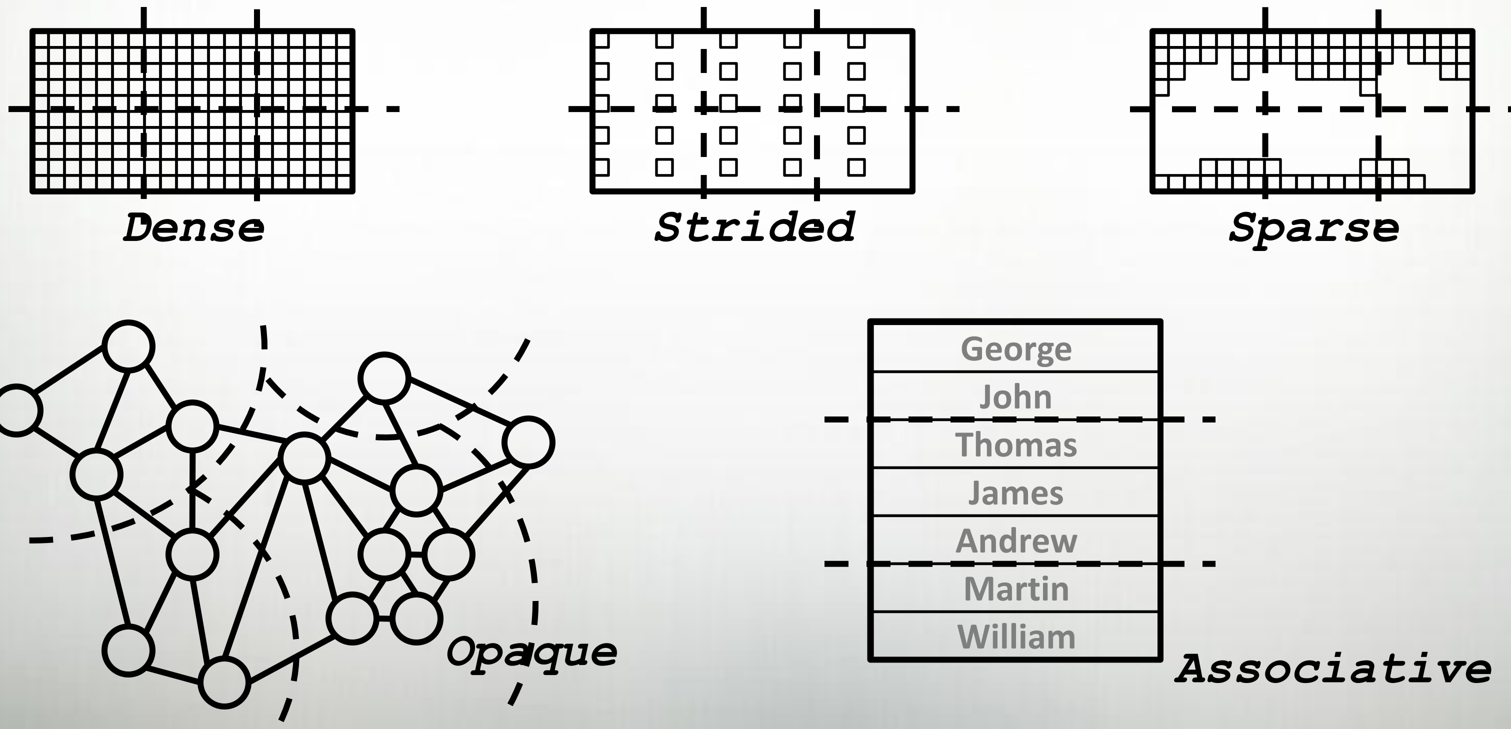
```
use myDMapMod;
var DMap: dmap(myDMap(...)) = new dmap(new myDMap(...));

var Dom: domain(...) dmapped DMap;
var A: [Dom] real;
```

Domain Map Types

All domain types can be dmapped.

Semantics are independent of domain map.

(Though performance and parallelism will vary...)



Outline

- Multi-Locale Basics
- Data Parallelism Revisited
- Domain Maps
- Chapel Standard Layouts and Distributions
  - Block
  - Cyclic
- User-defined Domain Maps

Chapel Standard Layouts and Distributions

Chapel provides a number of standard layouts and distributions

- All are written in Chapel

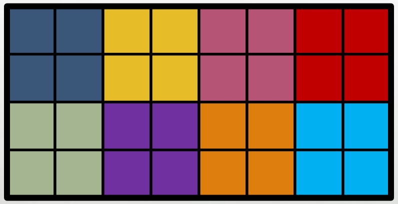
Examples

- Block distribution
- Cyclic distribution

The Block Distribution

The Block Distribution maps the indices of a domain in a dense fashion across the target Locales according to the `boundingBox` argument

```
const Dist = new dmap(new Block(boundingBox=[1..4, 1..8]));

var Dom: domain(2) dmapped Dist = [1..4, 1..8];
```



distributed over

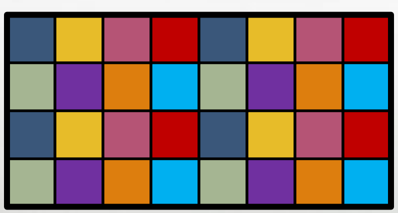
The Block class constructor

```
def Block(boundingBox: domain,
          targetLocales: [] locale = Locales,
          dataParTasksPerLocale = ...,
          dataParIgnoreRunningTasks = ...,
          dataParMinGranularity = ...,
          param rank = boundingBox.rank,
          type idxType = boundingBox.dim(1).eltType)
```

The Cyclic Distribution

The Cyclic Distribution maps the indices of a domain in a round-robin fashion across the target Locales according to the `startIdx` argument

```
const Dist = new dmap(new Cyclic(startIdx=(1,1)));

var Dom: domain(2) dmapped Dist = [1..4, 1..8];
```



distributed over

The Cyclic class constructor

```
def Cyclic(startIdx,
           targetLocales: [] locale = Locales,
           dataParTasksPerLocale = ...,
           dataParIgnoreRunningTasks = ...,
           dataParMinGranularity = ...,
           param rank: int = infered from startIdx,
           type idxType = infered from startIdx)
```

Distributions Example

- examples/primers/distributions.chpl

Outline

- Multi-Locale Basics
- Data Parallelism Revisited
- Domain Maps
- Chapel Standard Layouts and Distributions
- User-defined Domain Maps

User-defined Domain Maps

(Advanced) programmers can write domain maps

- The compiler uses a structural interface to build domain maps:
  - Create domains and arrays
  - Map indices to locales
  - Access array elements
  - Iterate over indices/elements sequentially, in parallel, zippered
  - ...

Standard Domain Maps ***are*** user-defined domain maps

Design goal: User-defined domain maps should perform as well as the Chapel Standard Domain Maps

Future Directions

- Heterogeneous locales
- Hierarchical locales
- GPU support via locales
- More standard distributions and layouts
- Specify interface for user-defined domain maps

Questions?

- Multi-Locale Basics
  - Locales
  - On, here, local, and communication
- Data Parallelism Revisited
- Domain maps
  - Layouts
  - Distributions
- The Chapel Standard Distributions
  - Block Distribution
  - Cyclic Distribution
- User-defined Domain Maps