

Chapel: Locality Control



The Locale

- **Definition**

- Abstract unit of target architecture
- Capable of running tasks and storing variables
 - i.e., has processors and memory
- Supports reasoning about locality

- **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

Locales and Program Startup

- Specify # of locales when running Chapel programs

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

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

numLocales: 8

LocaleSpace:



Locales:



- Chapel provides built-in locale variables

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

- main() begins as a single task on locale #0 (**Locales**[0])

Rearranging Locales

Create locale views with standard array operations:

```
var TaskALocs = Locales[0..1];  
var TaskBLocs = Locales[2..numLocales-1];  
var Grid2D = Locales.reshape([1..2, 1..4]);
```

Locales:

L0	L1	L2	L3	L4	L5	L6	L7
----	----	----	----	----	----	----	----

TaskALocs:

L0	L1
----	----

TaskBLocs:

L2	L3	L4	L5	L6	L7
----	----	----	----	----	----

Grid2D:

L0	L1	L2	L3
L4	L5	L6	L7

Locale Methods

- `def locale.id: int { ... }`

Returns locale's index in LocaleSpace

- `def locale.name: string { ... }`

Returns name of locale, if available (like uname -a)

- `def locale.numCores: int { ... }`

Returns number of processor cores available to locale

- `def locale.physicalMemory(...) { ... }`

Returns physical memory available to user programs on locale

Example

```
const totalPhysicalMemory =
  + reduce Llocales.physicalMemory();
```

The On Statement

- Syntax

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

- Semantics

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

- Examples

```
writeln("start on locale 0");
on Locales(1) do
  writeln("now on locale 1");
writeln("on locale 0 again");
```

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

SPMD Programming in Chapel Revisited

- A language may support both global- and local-view programming — in particular, Chapel does

```
def main() {
    forall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
}

def MySPMDProgram(me, p) {
    ...
}
```

Querying a Variable's Locale

- Syntax

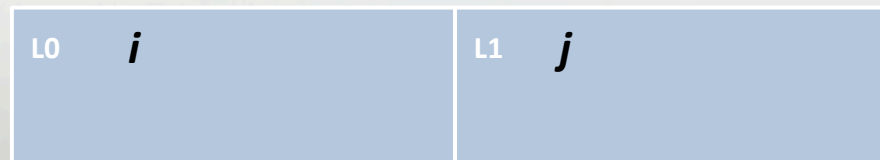
```
locale-query-expr:  
  expr . locale
```

- 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 value

```
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

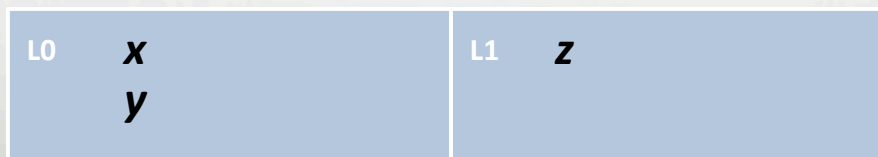
on Locales(1) {           // migrate task to locale 1
  var z: real;             // z allocated on locale 1

  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
  
```



Local statement

- Syntax

```
local-stmt:
  local { stmt };
```

- Semantics

- Asserts to the compiler that all operations are local

- Example

```
on Locales(1) {
  var x: int;
  local {
    x = here.id;
  }
  writeln(x); // outputs 1
}
```

Serial Example revisited

```

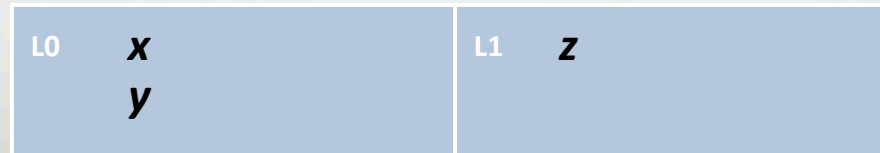
var x, y: real;           // x and y allocated on locale 0

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) {          // migrate back to locale 0
    var tz: real;
    local tz = x+y;         // no "checks" performed
    z = tz;                 // remote write to z
  }                          // migrate back to locale 1
  ...
}                            // migrate back to locale 0

```



Executing Multi-Locale Programs

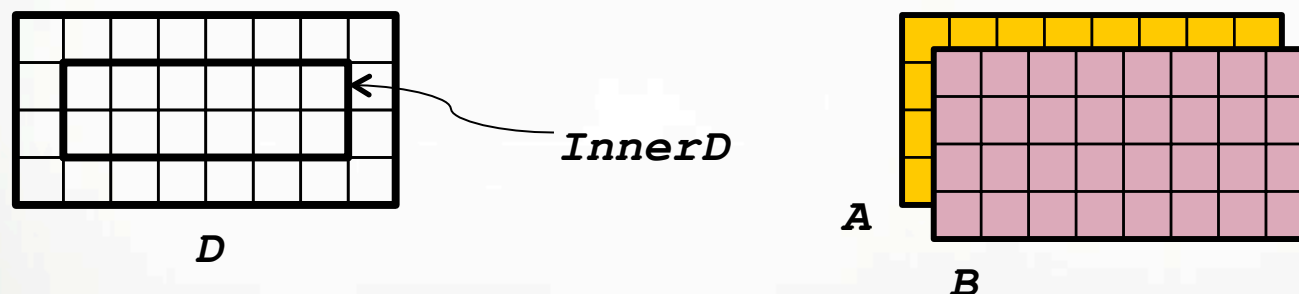
- By default, Chapel compiles for a single locale
 - environment variable CHPL_COMM defaults to 'none'
 - Effect: no communication inserted by compiler
 - Locales array supported, but has just one element
- To execute using multiple locales...
 - Set environment variable CHPL_COMM to 'gasnet'
 - (recompile Chapel runtime libraries)
 - See README.multilocale and README.launcher for further details

Outline

- Locales
- Domain Maps
 - Layouts
 - Distributions
- Chapel Standard Layouts and Distributions
- User-defined Domain Maps

Flashback: Data Parallelism

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



Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?

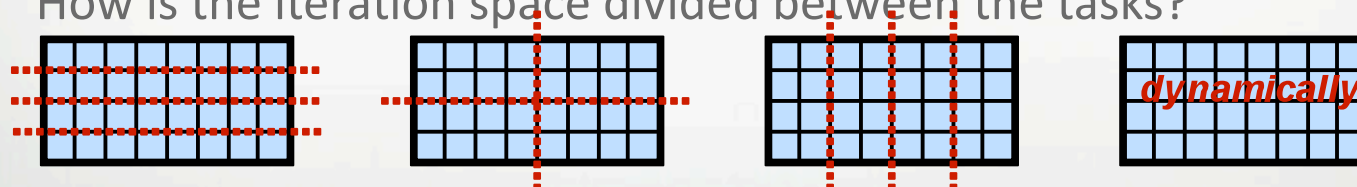
- Are regular arrays laid out in row- or column-major order? Or...?



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

Q2: How are data parallel operators implemented?

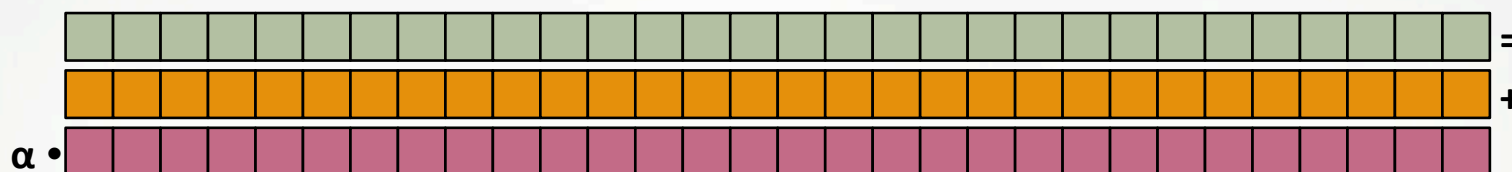
- How many tasks?
- How is the iteration space divided between the tasks?



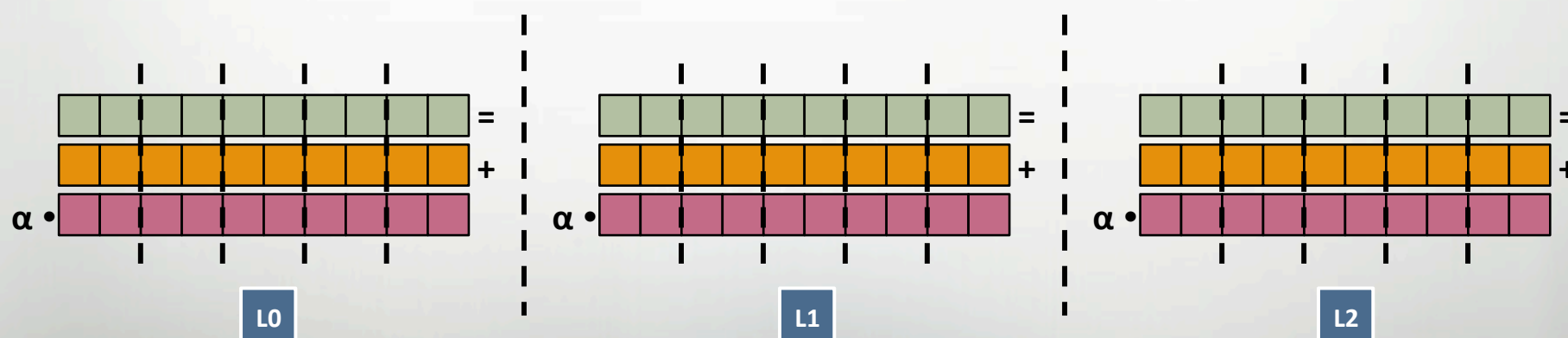
A: Chapel's *domain maps* are designed to give the user full control over such decisions

Domain Maps

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



...to a locale's memory and processors:



Domain Map Definitions

Domain maps define:

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

Domain maps are built using Chapel concepts

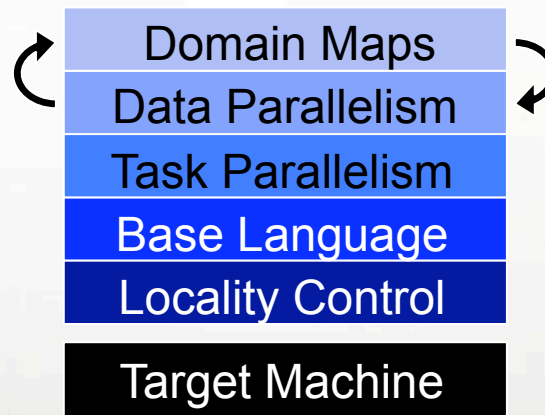
- classes, iterators, type inference, generic types
- task parallelism
- locales and on-clauses
- domains and arrays

Multiresolution Language Design, Revisited

Multiresolution Design: Support multiple tiers of features

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

Chapel language concepts



- separate concerns appropriately for clean design

Domain Maps: Layouts and Distributions

Domain Maps fall into two major categories:

layouts: target a single shared memory segment

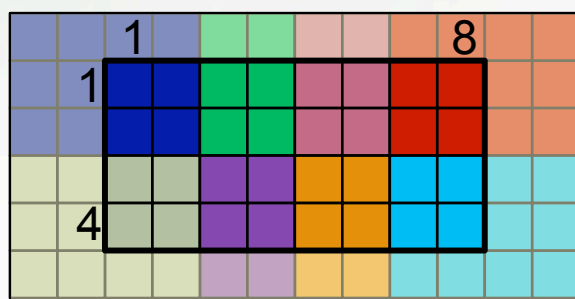
- (that is, a desktop machine or multicore node)
- **examples:** row- and column-major order, tilings, compressed sparse row

distributions: target distinct memory segments

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

Sample Distributions: Block and Cyclic

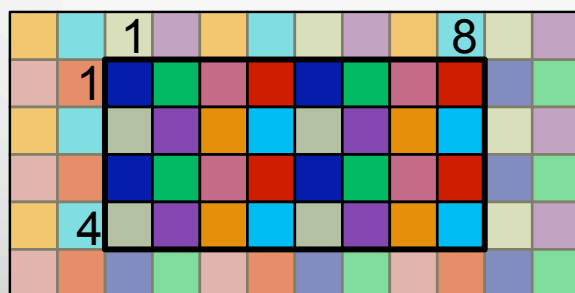
```
var Dom: domain(2) dmapped Block(boundingBox=[1..4, 1..8])  
      = [1..4, 1..8];
```



distributed to



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



distributed to



Chapel's Domain Map Strategy

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

Using Domain Maps

- Syntax

```

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

- Semantics

- Domain maps specify how a domain and its arrays are implemented

- Examples

```

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

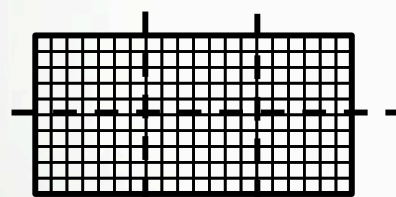
var Dom: domain (...) dmapped DMap;
var A: [Dom] real;
  
```

Domain Map Types

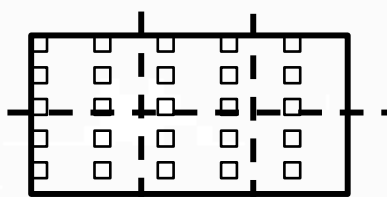
All domain types can be dmapped.

Semantics are independent of domain map.

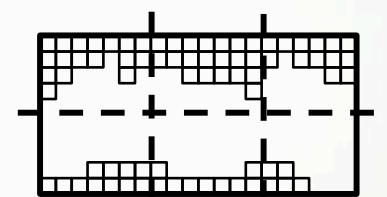
(Though performance and parallelism will vary...)



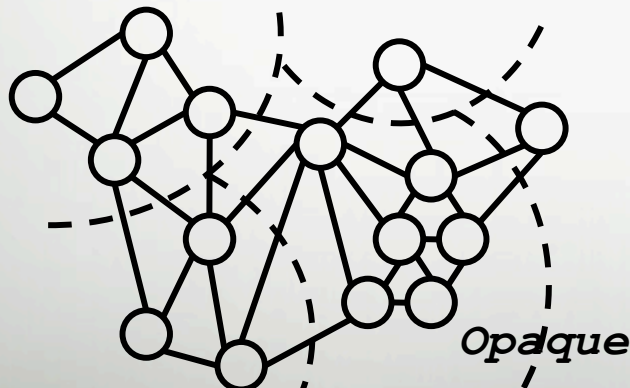
Dense



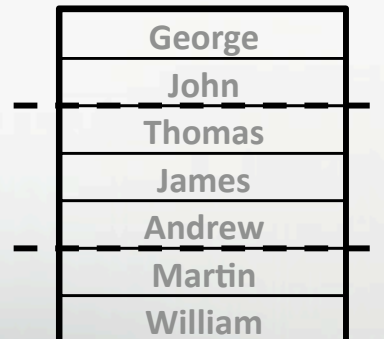
Strided



Sparse



Opaque



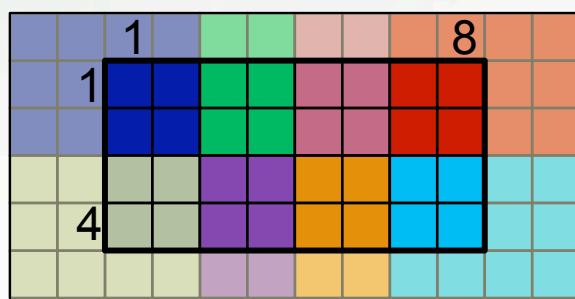
Associative

Outline

- Locales
- Domain Maps
- Chapel Standard Layouts and Distributions
 - Block
 - Cyclic
- User-defined Domain Maps

Sample Distributions: Block and Cyclic

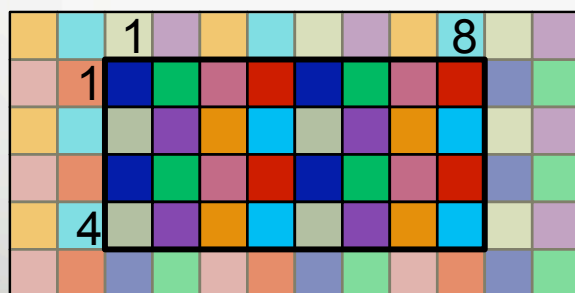
```
var Dom: domain(2) dmapped Block(boundingBox=[1..4, 1..8])
    = [1..4, 1..8];
```



distributed to



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

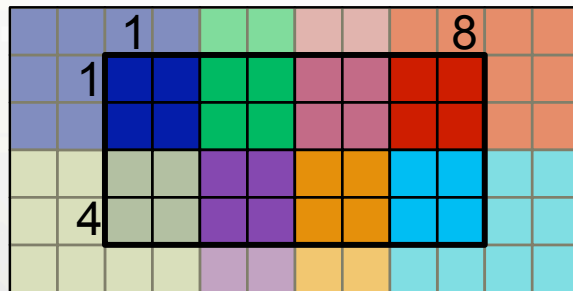


distributed to



The Block class constructor

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

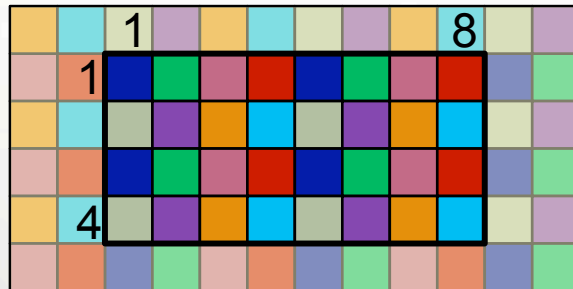


distributed to



The Cyclic class constructor

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



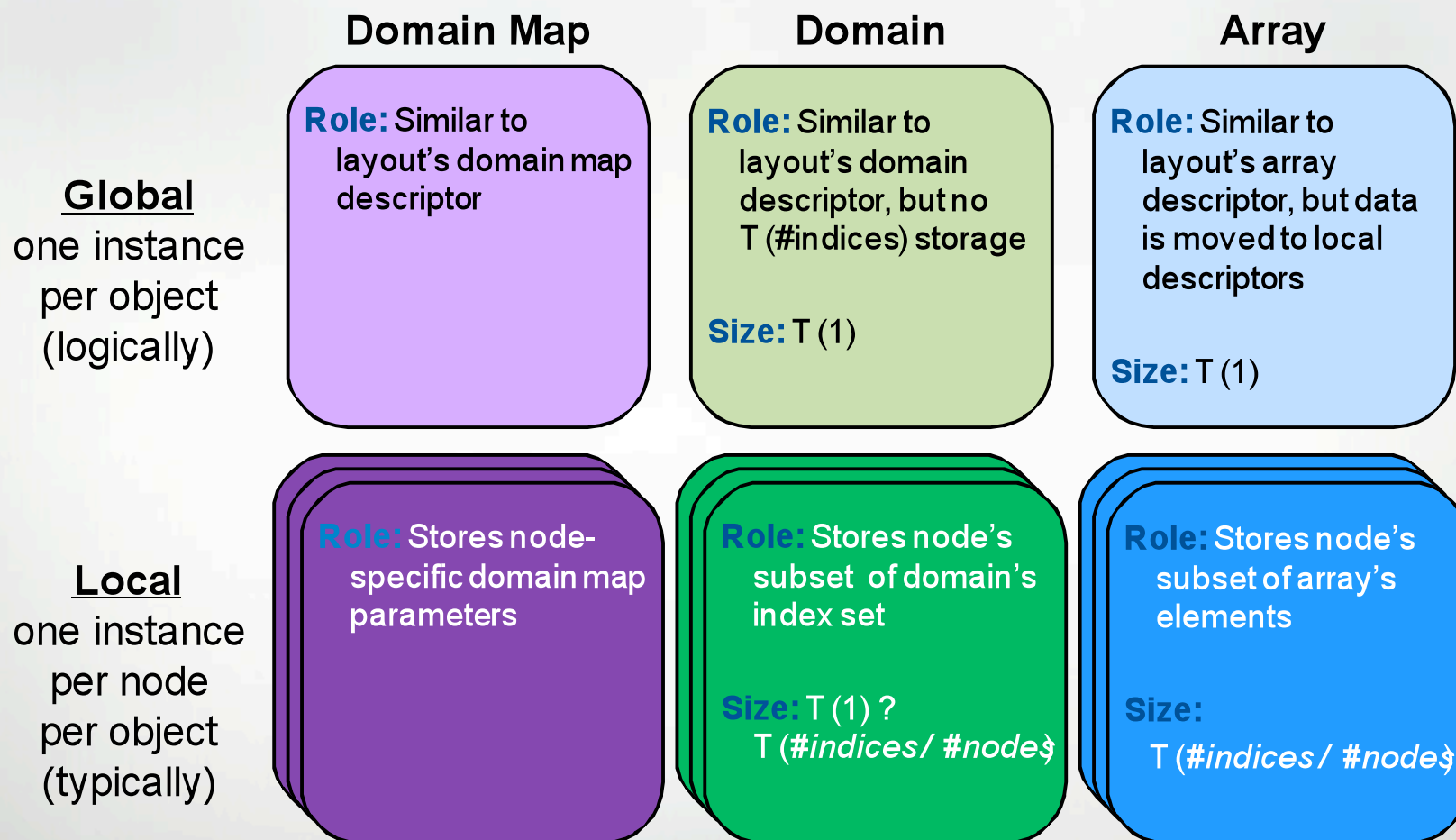
distributed to



Outline

- Locales
- Domain Maps
- Chapel Standard Layouts and Distributions
- User-defined Domain Map Descriptors

User-Defined Distribution Descriptors



Status: Locality

- Locales/on-clauses should be functioning perfectly
- Full-featured Block and Cyclic distributions
- Parallel sparse and associative layouts supported
- The compiler is currently conservative about assuming variables may be non-local
- Block-Cyclic, Associative distributions underway
- The compiler currently lacks several important communication optimizations
- Need to finalize user-defined domain map interfaces
- Need sparse and opaque distributions

Future Directions

- Hierarchical Locales
 - Expose hierarchy, heterogeneity within locales
 - Particularly important in next-generation nodes
 - CPU+GPU hybrids, tiled processors, manycore, ...
- Specify interface for user-defined domain maps
- Advanced uses of domain maps:
 - GPU programming
 - Dynamic load balancing
 - Resilient computation
 - *in situ* interoperability
 - Out-of-core computations

Questions?

- Multi-Locale Basics
 - Locales
 - on
- Domain maps
 - Layouts
 - Distributions
- The Chapel Standard Distributions
 - Block Distribution
 - Cyclic Distribution
- User-defined Domain Maps