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: [L0 L1 L2 L3 L4 L5 L6 L7]
Locales: [Locales[0] L1 L2 L3 L4 L5 L6 L7]
```

• 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])
Create locale views with standard array operations:

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

<table>
<thead>
<tr>
<th>Locales:</th>
<th>L0</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
<th>L7</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskALocs:</td>
<td>L0</td>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskBLocs:</td>
<td>L2</td>
<td>L3</td>
<td>L4</td>
<td>L5</td>
<td>L6</td>
<td>L7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid2D:</td>
<td>L0</td>
<td>L1</td>
<td>L2</td>
<td>L3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L4</td>
<td>L5</td>
<td>L6</td>
<td>L7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```chapel
const totalPhysicalMemory =
  + reduce Locales.physicalMemory();
```
The On Statement

- **Syntax**
  ```plaintext
  on-stmt:
    on expr { stmt }
  ```

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

- **Examples**
  ```plaintext
  writeln("start on locale 0");
  on Locales(1) do
    writeln("now on locale 1");
  writeln("on locale 0 again");
  ```

```plaintext
  var A: [LocaleSpace] real;
  coforall loc in Locales do
    on loc do
      A(loc.id) = compute(loc.id);
```
A language may support both global- and local-view programming — in particular, Chapel does

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

def MySPMDProgram(me, p) {
    ...
}
```
Querying a Variable's Locale

- **Syntax**
  
  \[
  \text{locale-query-expr:}
  \]
  
  \[
  \text{expr . locale}
  \]

- **Semantics**
  
  Returns the locale on which \textit{expr} is stored

- **Example**

  ```chapel
  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

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

on Locales(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
}
```

<table>
<thead>
<tr>
<th>L0</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>
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

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

on Locales(1) {
  var z: real; // z allocated on locale 1
  z = x + y; // remote reads of x and y
}

on Locales(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
```

<table>
<thead>
<tr>
<th>L0</th>
<th>x</th>
<th>L1</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.multilocal 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.
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 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 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

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

• separate concerns appropriately for clean design
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];
```

```
var Dom: domain(2) dmapped Cyclic(startIdx=(1,1)) = [1..4, 1..8];
```
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**
  
  \[
  \text{dmap-type: } \\
  \quad \text{dmap}(\text{dmap-class}(...)) \\
  \text{dmap-value: } \\
  \quad \text{new dmap(new dmap-class(...))}
  \]

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

- **Examples**
  
  ```chapel
  use myDMapMod;
  var DMap: dmap(myDMap(...)) = new dmap(new myDMap(...));
  
  var Dom: domain(...) dmapped DMap;
  var A: [Dom] real;
  ```
All domain types can be dmapped.
Semantics are independent of domain map.
(Though performance and parallelism will vary...)
Outline

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

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

![Block distribution diagram]

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

![Cyclic distribution diagram]
def Block(boundingBox: domain, 
    targetLocales: [] locale = Locales, 
    dataParTasksPerLocale = ..., 
    dataParIgnoreRunningTasks = ..., 
    dataParMinGranularity = ..., 
    param rank = boundingBox.rank, 
    type idxType = boundingBox.dim(1).eltType)
The Cyclic class constructor

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

```
distributed to
```

```
1 1 8
4
```

```
L0  L1  L2  L3
L4  L5  L6  L7
```
Outline

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

**Domain Map**
- **Role:** Similar to layout’s domain map descriptor
- **Global**
  - one instance per object (logically)
- **Local**
  - one instance per node per object (typically)
- **Size:** $\Theta(1)$ → $\Theta(\#\text{indices} / \#\text{nodes})$

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

**Array**
- **Role:** Similar to layout’s array descriptor, but data is moved to local descriptors
- **Size:** $\Theta(1)$

**Role:** Stores node-specific domain map parameters
- **Size:** $\Theta(1)$ → $\Theta(\#\text{indices} / \#\text{nodes})$

**Role:** Stores node’s subset of domain’s index set
- **Size:** $\Theta(\#\text{indices} / \#\text{nodes})$

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