Chapel: Locality

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

"Hello World" in Chapel: a Multi-Locale Version

- Multi-locale Hello World

```
coforall loc in Locales do
  on loc do
    writeln("Hello, world! ",
            "from node ", loc.id, " of ", numLocales);
```

Locales and Program Startup

- Specify # of locales when running Chapel programs

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

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

- Chapel provides built-in locale variables

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

numLocales: 8

LocaleSpace:

Locales:

L0	L1	L2	L3	L4	L5	L6	L7

- 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]);
```

Locale Methods

- ```
  proc locale.id: int { ... }
  ```
  Returns locale’s index in LocaleSpace
- ```
  proc locale.name: string { ... }
  ```
  Returns name of locale, if available (like uname -a)
- ```
  proc locale.numCores: int { ... }
  ```
  Returns number of processor cores available to locale
- ```
  proc 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*
- Example

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

Locality and Parallelism are Orthogonal

- On-clauses do not introduce any parallelism

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

- But can be combined with constructs that do:

```
writeln("start on locale 0");
begin on Locales(1) do
  writeln("now on locale 1");
on Locales(2) do begin
  writeln("now on locale 2");
writeln("on locale 0 again");
```

- (the final three statements could appear in any order)

SPMD Programming in Chapel Revisited

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

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

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

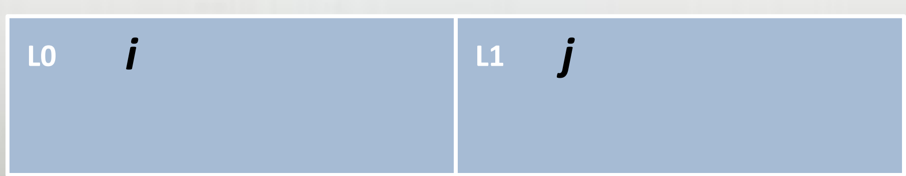
Querying a Variable's Locale

- 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
```

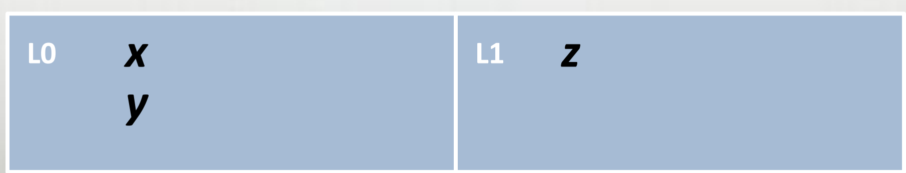
Serial Example with Implicit Communication

```
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
```



Status: Locales

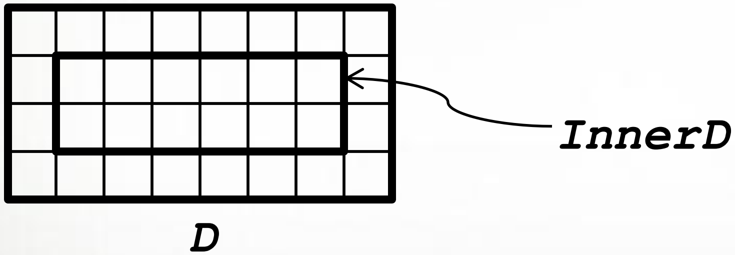
- Everything should be functioning perfectly
- The compiler is currently conservative about assuming variables may be remote
  - Impact: scalar performance overhead
- The compiler is currently lacking several important communication optimizations
  - Impact: performance impact for programs that would benefit by aggregated communication

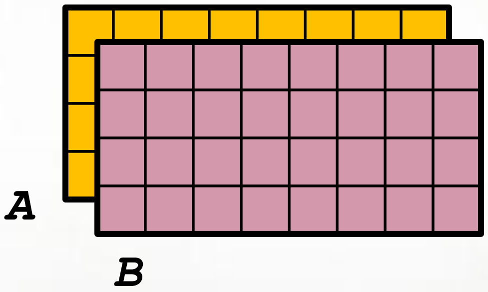
Future Directions

- Hierarchical Locales (joint work with UIUC)
  - Support ability to expose hierarchy, heterogeneity within locales
  - Particularly important in next-generation nodes
    - CPU+GPU hybrids
    - tiled processors
    - manycore processors

Review: Data Parallelism

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

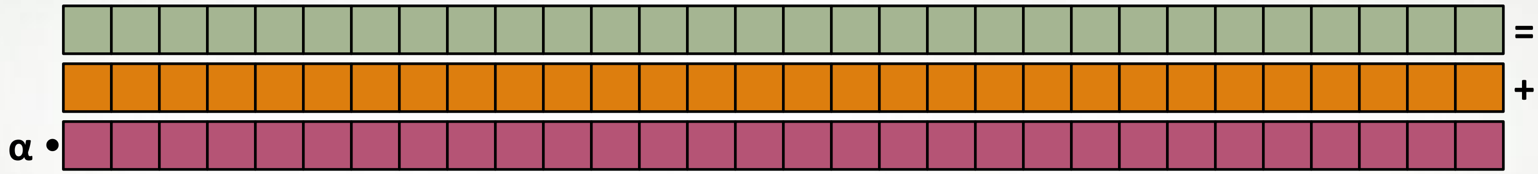




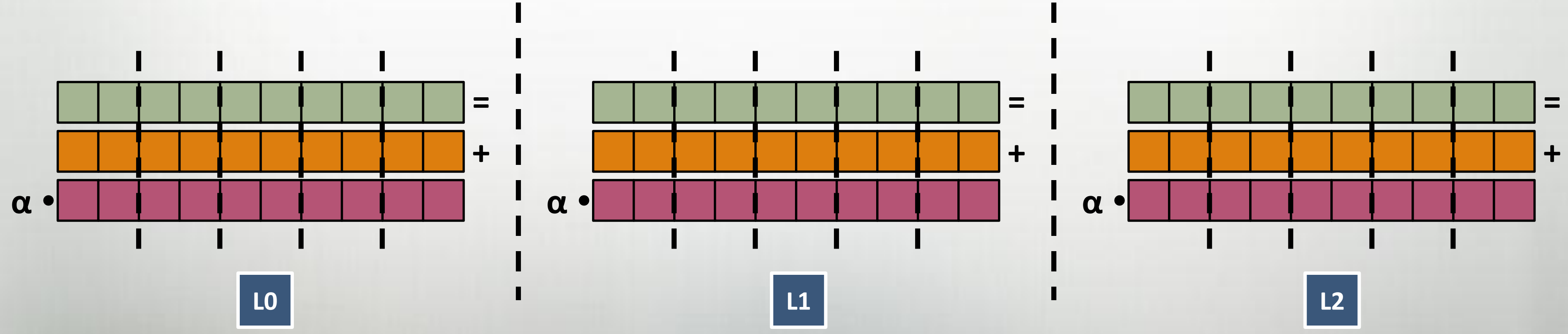
- May optionally be distributed over multiple locales
- Can be stored in local memories in arbitrary ways

Domain Maps

Domain maps are "recipes" that instruct the compiler how to map the global view of a computation…

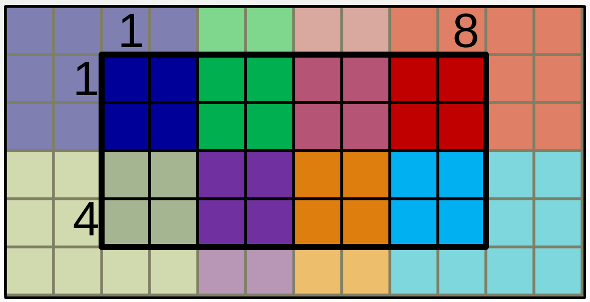


…to a locale's memory and processors:



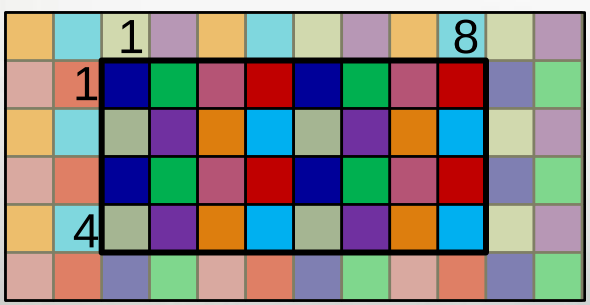
Sample Distributions: Block and Cyclic

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

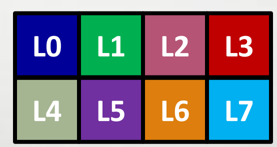


distributed to


```
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

Domain Map Details: IOU

- I will be talking at length about domain maps on Tuesday morning, so thought I'd save some time here by asking you to attend that talk

Domain Maps: Status

- Full-featured Block, Cyclic, and Replicated distributions
- Single-locale COO and CSR Sparse layouts supported
- Serial quadratic probing Associative layout supported
- Block-Cyclic, Dimensional, and Associative distributions underway
- Adding documentation for defining domain maps
- 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

Questions?