Chapel: Locales

(Controlling Locality and Affinity)
The Locale Type

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

Properties:
• a locale’s tasks have ~uniform access to local vars
• Other locale’s vars are accessible, but at a price

In practice:
• Typically a compute node (multicore processor or SMP)
Multi-locale Hello World

```chapel
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

  ```chapel
  config const numLocales: int = ...;
  const LocaleSpace = {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:

```plaintext
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..];
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

- **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**

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

• **Syntax**

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

• **Semantics**

  • Executes *stmt(s)* on the locale that stores *expr*

• **Example**

  ```
  writeln("start executing on locale 0");
  on Locales[1] do
    writeln("now we’re on locale 1");
  writeln("back on locale 0 again");
  ```
Locality and Parallelism are Orthogonal

- On-clauses do not introduce any parallelism

```chapel
writeln("start executing on locale 0");
on Locales[1] do
    writeln("now we're on locale 1");
writeln("back on locale 0 again");
```

- But can be combined with constructs that do:

```chapel
writeln("start executing on locale 0");
cobegin {
    on Locales[1] do
        writeln("this task runs on locale 1");
    on Locales[2] do
        writeln("while this one runs on locale 2");
}
writeln("back on locale 0 again");
```

- Orthogonal support for parallelism and locality is key
A language may support both global- and local-view programming — in particular, Chapel does

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

proc MySPMDProgram(me, p) {
    ...
}
```
Data-driven on-clauses

• On-clauses can also use a data-driven form...

```cobegin
  on node.left do
    search(node.left);
  on A[i,j] do
    bigComputation(A);
end```

...supporting affinity between tasks and their data

(Note that even the ‘on Locales[3]’ form can be considered data-driven, since each locale stores its respective locale value)
Q: How does data get onto other locales to begin with?

A1: Lexical scoping

```plaintext
var x: int; // x is stored on locale 0
on Locales[1] {
    var y: int; // y is stored on locale 1
    on Locales[2] {
        var z: int; // z is stored on locale 2

        on y { y -= 1; } // executes on locale 1
    }
}
```

<table>
<thead>
<tr>
<th>Loc 0</th>
<th>Loc 1</th>
<th>Loc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>
**Q:** How does data get onto other locales to begin with?

**A2:** Class instances

```java
class C { var x, y, z: real; var next: C; }

var myC: C; // myC is stored on locale 0

on Locales[1] { 
    myC = new C(...); // myC’s object lives on locale 1...
    on Locales[2] do
        myC.next = new C(...); // and its next is on locale 2
    }

on myC do ... // executes on locale 1
on myC.next do ... // executes on locale 2
```
Q: How does data get onto other locales to begin with?

A3: On-declarations (not yet implemented)

```
on Locales[1] var x: real;  // x is stored on locale 1
on Locales[2] var y: real;  // y is stored on locale 2
on x do ...  // executes on locale 1
on y do ...  // executes on locale 2
```
Q: How does data get onto other locales to begin with?

A4: Distributed domains and arrays (next slide deck)
Querying a Variable's Locale

- **Syntax**
  
  ```
  locale-query-expr:
      expr . locale
  ```

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

- **Example**

```chapel
var i: int;
on Locales[1] {
  var j: int;
  writeln((i.locale.id, j.locale.id)); // outputs (0,1)
}
```
• **Built-in locale variable**

```plaintext
const here: locale;
```

• **Semantics**

• Refers to the locale on which the task is executing

• **Example**

```plaintext
writeln(here.id); // outputs 0

on Locales[1] do
  writeln(here.id); // outputs 1

on myC do
  if (here == Locales[0]) then ...
```
The compiler can optimize communication subject to Chapel’s memory consistency model.

```plaintext
var x: int;

on Locales[1] { // on-clause implies an active message
  var y: int;
  y = x; // in practice, read-only values like x
  // are bundled with the active message
}
```
Local statement

- **Syntax**

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

- **Semantics**

  - Asserts to the compiler that all operations are local

- **Example**

  ```
  on Locales[1] { 
  var myC: C = ...; 
  ... 
  myC.x += 1; // is myC.x local? 
  }
  ```

- **Note**: Our current hope is to deprecate this feature, replacing it with data-centric concepts
Status: Locales

- Most everything works correctly
  - exception: the on-declaration syntactic form
- 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: scalability tends to be limited for programs with structured communication
Future Directions

• Hierarchical Locales (currently being developed)
  • Support ability to expose hierarchy, heterogeneity within locales
  • Particularly important in next-generation nodes
    • CPU+GPU hybrids
    • tiled processors
    • manycore processors
Questions?

• Multi-Locale Basics
  • locales
  • on-clauses
  • data placement
  • .locale
  • here
  • communication implications
  • local
Prototypical Next-Gen Processor Technologies

Intel MIC

AMD Trinity

Nvidia Echelon

Tilera Tile-Gx

General Characteristics of These Architectures

- Increased hierarchy and/or sensitivity to locality
- Potentially heterogeneous processor/memory types

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past
Locales Today

Concept:

- Today, Chapel supports a 1D array of locales
  - users can reshape/slice to suit their computation’s needs

- Apart from queries, no further visibility into locale structure
  - no mechanism to refer to specific NUMA domains, processors, memories, ...
  - assumption: compiler, runtime, OS, HW can handle intra-locale concerns
Current Work: Hierarchical Locales

Concept:

- Support locales within locales to describe architectural sub-structures within a node

<table>
<thead>
<tr>
<th>sub-locale A</th>
<th>sub-locale A</th>
<th>sub-locale A</th>
<th>sub-locale A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- As with traditional locales, *on-clauses* and *domain maps* can be used to map tasks and variables to a sub-locale’s memory and processors

- Locale structure is defined as Chapel code
  - permits implementation policies to be specified in-language
  - introduces a new Chapel role: *architectural modeler*