Chapel: Locales

(Controlling Locality and Affinity)
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
Multi-locale Hello World

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

"Hello World" in Chapel: a Multi-Locale Version
Specify # of locales when running Chapel programs

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

Chapel provides built-in locale variables

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

```
numLocales: 8
LocaleSpace: L0 L1 L2 L3 L4 L5 L6 L7
Locales: L0 L1 L2 L3 L4 L5 L6 L7
```

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]);
```

### 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 Locales.physicalMemory();
```
The On Statement

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

- **Semantics**
  - Executes `stmt` on the locale that stores `expr`

- **Example**
  
  ```chapel
  writeln("start on locale 0");
  on Locales(1) do
    writeln("now on locale 1");
  writeln("on locale 0 again");
  ```
• On-clauses do not introduce any parallelism

```chapel
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:

```chapel
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)
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**

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

• **Semantics**

- Returns the locale on which `expr` is stored

• **Example**

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

<table>
<thead>
<tr>
<th>L0</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here

• **Built-in locale value**

```chapel
const here: locale;
```

• **Semantics**
  - Refers to the locale on which the task is executing

• **Example**

```chapel
writeln(here.id);       // outputs 0
on Locales(1) do
  writeln(here.id);    // outputs 1
```
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
  $z = x + y$;  // remote write to $z$
  // migrate back to locale 1
on $x$ do
  $z = x + y$;  // remote write to $z$
  // migrate back to locale 1
}  // migrate back to locale 0
Local statement

- **Syntax**
  
  ```chapel
  local-stmt:
  local { stmt };
  ```

- **Semantics**
  - Asserts to the compiler that all operations are local

- **Example**
  
  ```chapel
  on Locales(1) {
    var x: int = ...;
    var y: int = ...;
    local {
      x += y;
    }
    writeln(x);  // outputs 1
  }
  ```
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
Everything should be functioning perfectly

The compiler is currently conservative about assuming variables may be non-local
  - 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 (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
Questions?

- Multi-Locale Basics
  - Locales
  - on
  - here
  - local