Locality / Affinity Features
What is a Locale?

Definition:

- Abstract unit of target architecture
- Supports reasoning about locality
  - defines “here vs. there” / “local vs. remote” / ”cheap vs. $$$”
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A compute node (multicore processor or SMP)
Getting started with locales

- Users 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 Locales: [0..#numLocales] locale = ...;
  ```

- User’s `main()` begins executing on locale #0
Locale Operations

- **Locale methods support queries about the target system:**

  ```cray
  proc locale.physicalMemory(...) { ... }
  proc locale.numPUs() { ... }
  proc locale.id { ... }
  proc locale.name { ... }
  ```

- **On-clauses support placement of computations:**

  ```cray
  writeln("on locale 0");
  on Locales[1] do
      writeln("now on locale 1");
  writeln("on locale 0 again");
  on A[i,j] do
      bigComputation(A);
  on node.left do
      search(node.left);
  ```
Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:

  ```chapel
  begin writeln("Hello world!");
  writeln("Goodbye!");
  ```

- This is a **distributed**, but serial program:

  ```chapel
  writeln("Hello from locale 0!");
  on Locales[1] do writeln("Hello from locale 1!");
  writeln("Goodbye from locale 0!");
  ```

- This is a **distributed** and **parallel** program:

  ```chapel
  begin on Locales[1] do writeln("Hello from locale 1!");
  on Locales[2] do begin writeln("Hello from locale 2!");
  writeln("Goodbye from locale 0!");
  ```
Partitioned Global Address Space (PGAS) Languages

(Or perhaps: partitioned global namespace languages)

● abstract concept:
  ● support a shared namespace on distributed memory
    ● permit parallel tasks to access remote variables by naming them
  ● establish a strong sense of ownership
    ● every variable has a well-defined location
    ● local variables are cheaper to access than remote ones

● traditional PGAS languages have been SPMD in nature
  ● best-known examples: Fortran Co-Arrays, UPC

partitioned shared name-/address space

private space 0  private space 1  private space 2  private space 3  private space 4

Copyright 2018 Cray Inc.
SPMD PGAS Languages (using a pseudo-language, not Chapel)

shared int i(*); // declare a shared variable i
shared int i(*); // declare a shared variable i
function main() {
    i = 2*this_image(); // each image initializes its copy
SPMD PGAS Languages (using a pseudo-language, not Chapel)

shared int i(*); // declare a shared variable i
function main() {
    i = 2*this_image(); // each image initializes its copy
}

private int j; // declare a private variable j
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```plaintext
shared int i(*); // declare a shared variable i

function main() {
    i = 2*this_image(); // each image initializes its copy
    barrier();

    private int j; // declare a private variable j

    j = i((this_image()+1) % num_images()); // ^^ access our neighbor’s copy of i

    // communication is implemented by the compiler + runtime

    // Q: How did we know our neighbor had an i? A: Because it’s SPMD – we’re
    // all running the same program. (Simple, but restrictive)
```

```
<table>
<thead>
<tr>
<th>i=</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Chapel is PGAS, but unlike most, it’s not inherently SPMD
- never think about “the other copies of the program”
- “global name/address space” comes from lexical scoping
  - as in traditional languages, each declaration yields one variable
  - variables are stored on the locale where the task declaring it is executing

Locales (think: “compute nodes”)
Chapel: Scoping and Locality

var i: int;

Locales (think: “compute nodes”)
Chapel: Scoping and Locality

```chapel
var i: int;
on Locales[1] {
    "Locales (think: “compute nodes”)"
}
```
Chapel: Scoping and Locality

```plaintext
var i: int;
on Locales[1] {
    var j: int;
}
```

Locales (think: “compute nodes”)

Copyright 2018 Cray Inc.
Chapel: Scoping and Locality

```chapel
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
```
Chapel: Scoping and Locality

```chapel
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
            var k: int;
            ...
        }
    }
}
```

`Locales` (think: “compute nodes”)
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = 2*i + j;
    }
  }
}

OK to access i, j, and k wherever they live

Locales (think: “compute nodes”)

k = 2*i + j;
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
            var k: int;
            k = 2*i + j;
        }
    }
}

here, i and j are remote, so the compiler + runtime will transfer their values

Locales (think: “compute nodes”)
Chapel: Locality queries

```chapel
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
            var k: int;

            ...here...  // query the locale on which this task is running
            ...j.locale...  // query the locale on which j is stored
        }
    }
}
```

Locales (think: “compute nodes”)
Chapel: Locality queries

```chapel
definition Locales = Locality (think: "compute nodes")

definition here = Local (think: "compute node")

definition j.locale = Local (think: "compute node")

var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
            var k: int;

            ...here...
            ...j.locale...
        }
    }
}
```

Recall: scalars have ‘const in’ task intents by default.

Locales (think: “compute nodes”)

// query the locale on which this task is running
// query the locale on which i is stored

```plaintext
here.id = 3
j.locale.id = 3
```

```plaintext
i: 0
j: 1
k: 2
k: 3
k: 4
```
Chapel: Locality queries

```chapel
cvar i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales with (ref i, ref j) {
        on loc {
            var k: int;

            ...here...
            // query the locale on which this task is running
            ...j.locale...
            // query the locale on which i is stored
        }
    }
}
```

Locales (think: “compute nodes”)
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 (0,1)
  }
  ```
Here

- **Built-in locale variable**
  
  ```
  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
  on myC do
    if (here == Locales[0]) then ...
  ```
Though implicit, users can reason about communication

- semantic model is explicit about where data is placed / tasks execute
- execution-time queries support reasoning about locality
  - e.g., `here`, `x.locale`
- tools should also play a role here
  - e.g., `CommDiagnostics` module which can count communications
  - e.g., `chplvis`, contained in the release (developed by Phil Nelson, WWU)
Rearranging Locales

Create locale views with standard array operations:

```plaintext
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..];
var Grid2D = reshape(Locales, {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
```
Questions about (low-level) locality in Chapel?
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publicly announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.