Chapel: Locality and Affinity

Sung-Eun Choi and Steve Deitz
Cray Inc.
Outline

• Multi-Locale Basics
  • Locales
  • On, here, local, and communication
• Optimizations
• Fragmented execution model subsumed
The Locale Type

• Definition
  • Abstract unit of target architecture
  • Capacity for processing and storage
  • Supports reasoning about locality

• Properties
  • Locale’s tasks have uniform access to local memory
  • Other locale’s memory is accessible, but at a price

• Examples
  • A multi-core processor
  • An SMP node
Program Startup

• Execution Context

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

• Specify # of locales when running executable

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

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

• Execution begins as a single task on a locale 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]);
```
Locale Methods

- **def locale.id: int { ... }**
  Returns index in LocaleSpace

- **def locale.name: string { ... }**
  Returns name of locale (like `uname -a`)

- **def locale.numCores: int { ... }**
  Returns number of 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**
  
  ```chapel
  on-stmt:
    on expr { stmt }
  ```

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

- **Example**
  
  ```chapel
  var A: [LocaleSpace] int;
  coforall loc in Locales do
    on loc do
      A(loc.id) = compute(loc.id);
  ```
Querying a Variable's Locale

- **Syntax**

  ```chapel
  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 01
  }
  ```

  ![Diagram of variables `i` and `j` across locales L0 and L1]
Here

- Built-in locale
  ```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) {  // 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

L0  x  y
L1  z
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;
  local {
      x = here.id;
  }
  writeln(x);  // outputs 1
  }
  ```
`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) {`  // migrate back to locale 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
Outline

- Multi-Locale Basics
- Optimizations
  - Eliminating local task creation
  - Remote value forwarding
- Fragmented execution model subsumed
Eliminating local task creation

- **Example**

```chapel
begin on Locales(1) {
    writeln(here.id);
}
```

- **Becomes..**
• Applies to cobegin and coforall statements too

```chapel
coforall loc in Locales do on loc do
  writeln(\texttt{here}.id);
```

Becomes...

[Diagram showing the execution flow between levels]

---

Chapel: Locality and Affinity
Remote value forwarding

- Example

```chapel
var x: real; //allocated on L0
on Locales(1) {
    writeln(x);
}
```

- Becomes...

```chapel
on with x {
    get x;
    writeln(x);
}
```
**Serial Example revisited again**

```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; // local 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>L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>
Outline

- Multi-Locale Basics
- Optimizations
- Fragmented execution model subsumed
def main() {
    coforall loc in Locales do on loc {
        myFragmentedMain();
    }
}

def myFragmentedMain() {
    const size = numLocales, rank = here.id;
    ...
}
Future Directions

- Heterogeneous locales
- Hierarchical locales
- GPU support via locales
Questions?

- Multi-Locale Basics
  - Locales
  - On, here, local, and communication
- Optimizations
  - Eliminating local task creation
  - Remote value forwarding
- Fragmented execution model subsumed