Chapel: Locality and Affinity

Steve Deitz

Outline

- Basics of Multi-Locale Chapel
  - The `locale` type and `Locales` array
  - The `on` statement, `here` locale, and communication
  - The `local` block and `ubiq` variables
- `MyDistributedArray` Example
- Array and Domain Distributions
The locale Type

- **Definition**
  - An architectural unit of locality
  - Has capacity for processing and storage
- **Properties**
  - Threads within a locale have ~uniform access to local memory
  - Memory within other locales is accessible, but at a price
- **Example**
  - A multicore processor or SMP node could be a locale

Chapel Programming Model

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

- **Explicit Parallelism and Locality Model**
  - Execution on remote locales is introduced by `on`
  - Parallelism is introduced by `begin/cobegin/coforall`
  - Note: Distributions may employ the above constructs

- **Starting a program**
  - Execution begins with one task running on Locale 0
  - The number of locales is specified on the command line
    ```sh
    > a.out -nl 2
    ```
Executing on Remote Locales

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

- **Semantics**
  - Executes the statement on the locale specified by the expression
  - Does not introduce concurrency

- **Example**
  ```
  var A: [LocaleSpace] int;
  coforall loc in Locales do on loc {
  A(loc.id) = computation(loc.id);
  }
  ```

  - Note: `locale.id` returns a locale’s index in the Locales array

Here

- **Built-in locale**
  ```
  const here: locale;
  ```

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

- **Example**
  ```
  writeln(here.id);
  on Locales(1) do
  writeln(here.id);
  ```

- **Output**
  ```
  0
  1
  ```
Querying a Locale

- Syntax

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

- Semantics
  - Evaluates the locale on which the expression is located

- Example

```chapel
var i: int;
on Locales(1) {
  write(i.locale.id);
on i do write(here.id);
}
```

- Output

```
00
```

Remote Reads and Writes

- Example

```chapel
var i = 0;
on Locales(1) {
  writeln((here.id, i.locale.id, i));
i = 1;
  writeln((here.id, i.locale.id, i));
} writeln((here.id, i.locale.id, i));
```

- Output

```
(1, 0, 0)
(1, 0, 1)
(0, 0, 1)
```
Remote Classes

- Example

```chapel
class C {
    var x: int;
}
var c: C;
on Locales(1) do c = new C();
writeln((here.id, c.locale.id, c));
```

- Output

```
(0, 1, {x = 0})
```

Local Blocks

- Syntax

```
local stmt:
    local stmt
```

- Semantics

  - Asserts there is no communication in the local statement
  - Runtime checks can be disabled

- Example

```chapel
c = Root.child(1);
on c do local {
    traverseTree(c);
}
local {
    A(D) = B(D);
}
```
Ubiquitous Variables

- Syntax
  
  \[
  \text{ubiquitous-variable-declaration:}
  \]
  
  \[
  \text{ubiq variable-declaration}
  \]

- Semantics
  - Each locale has its own copy of this variable
  - Can be used to replicate data

- Example
  
  \[
  \text{ubiq var } i: \text{ int;}
  \]
  
  \[
  \text{for loc in Locales do on loc {}
  \]
  
  \[
  i = \text{loc.id;}
  \]
  
  \[
  }
  \]

- Note: \textbf{here} is a ubiquitous constant of locale type

Outline

- Basics of Multi-Locale Chapel
- MyDistributedArray Example
  - Or how to build a class that acts like a simple distributed array
- Array and Domain Distributions
MyDistributedArray Class Declaration

```plaintext
class MyDistributedArray {
    var myInds: domain(1);
    var myData: [myInds] int;
    var others: [LocaleSpace] MyDistributedArray;
}
```

MyDistributedArray Access Function

```plaintext
def MyDistributedArray.this(i: int) var {
    if myInds.member(i) then
        return myData(i);
    else
        return others((i-1)*numLocales/n).myData(i);
}
```
MyDistributedArray Construction

```chapel
var A: MyDistributedArray;
var AS: [LocaleSpace] MyDistributedArray;

for loc in Locales do on loc do
    AS(loc.id) = new MyDistributedArray([loc.id*n/numLocales+1..
                                           (loc.id+1)*n/numLocales]);

A = AS(0);

for loc in Locales do
    AS(loc.id).others = AS;

for i in 1..n do // A is a hotspot
    A(i) = i;
```

Applying Ubiq to MyDistributedArray

```chapel
ubiq var A: MyDistributedArray;

var AS: [LocaleSpace] MyDistributedArray;

for loc in Locales do on loc {
    AS(loc.id) = new MyDistributedArray([loc.id*n/numLocales+1..
                                           (loc.id+1)*n/numLocales]);
    A = AS(loc.id);
}

for loc in Locales do
    AS(loc.id).others = AS;

for i in 1..n do // No more hotspot
    A(i) = i;
```
Outline

- Basics of Multi-Locale Chapel
- MyDistributedArray Example
- Array and Domain Distributions
  - The distribution—domain—array hierarchy
  - Using distributions

What is a Distribution?

A distribution is a structure that implements…
...the mapping from indices to locales
...the per-locale representation of domain indices and array elements
...the compiler’s target interface for lowering global-view operations
What is a Distribution?

A distribution is a structure that implements...

...the mapping from indices to locales
...the per-locale representation of domain indices and array elements
...the compiler's target interface for lowering global-view operations

Distributing Domains

Domains may be distributed across locales

```chapel
var Dist = new Block();
var D: domain(2) dist Dist;
var A,B: [D] real;
```

A distribution implies...

...ownership of the domain’s indices (and its arrays' elements)
...the default work ownership for operations on the domains/arrays
Distributions (Work in Progress)

- Distributions support lowering
  - From the user's global view operations on a distributed array
  - To the fragmented implementation for a distributed memory machine

- Users can implement custom distributions
  - Using task parallel features, on clauses, domains/arrays
  - Must implement standard interface
    - For allocation/reallocation of domain indices and array elements
    - For mapping indices to locales and values
    - For iterating in parallel and serial
    - ...
  - Optional interface for performance

- Chapel provides a standard library of distributions...
  - Uses the same mechanism as user-defined distributions
  - Tuned for different platforms to maximize performance

Chapel Distributions

<table>
<thead>
<tr>
<th></th>
<th>distribution</th>
<th>domain</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global descriptors</strong></td>
<td>Responsibility: Mapping of indices to locales</td>
<td>Responsibility: How to store, iterate over domain indices</td>
<td>Responsibility: How to store, access, iterate over array elements</td>
</tr>
<tr>
<td>(one global instance or replicated per locale)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>local descriptors</strong></td>
<td>Responsibility: How to store, iterate over local domain indices</td>
<td>Responsibility: How to store, access, iterate over local array elements</td>
<td></td>
</tr>
<tr>
<td>(one instance per locale)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapel Distributions

- **global descriptors** (one global instance or replicated per locale)
  - target locale set
  - distribution params
  - map index to locale

- **local descriptors** (one instance per locale)
  - store local indices
  - local index iteration
  - add new indices

Legend:
- = descriptor state
= descriptor methods

Questions?