Chapel: Locality and Affinity

Brad Chamberlain

Outline

- Basics of Multi-Locale Chapel
  - The `locale` type and `Locales` array
  - The `on` statement, `here` locale, and communication
  - The `local` block
- Domain and Array Distributions
- Sample Uses of Distributed Domains/Arrays
The locale Type

- **Definition**
  - An abstract unit of the target architecture
  - Supported to permit reasoning about 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
  - Locales are defined for a given architecture by a Chapel compiler
    - e.g., a multicore processor or SMP node could be a locale

Locales and Program Startup

- **Chapel users specify # locales on executable command-line**
  
  `prompt> myChapelProg -nl=8   # run using 8 locales
  `L0 L1 L2 L3 L4 L5 L6 L7`

- **Chapel launcher bootstraps program execution:**
  - obtains necessary machine resources
    - *e.g.*, requests 8 nodes from the job scheduler
  - loads a copy of the executable onto the machine resources
  - starts running the program. Conceptually…
    - …locale #0 starts running program’s entry point (`main()`)
    - …other locales wait for work to arrive
Locale Variables

Built-in variables represent a program’s set of locales:

```chapel
config const numLocales: int;         // number of locales
const LocaleSpace = [0..numLocales-1]; // locale indices
Locales: [LocaleSpace] locale;        // locale values
```

```
numLocales:  8

LocaleSpace: 0 1 2 3 4 5 6 7

Locales:  L0 L1 L2 L3 L4 L5 L6 L7
```

Locale Views

Using standard array operations, users can create their own locale views:

```chapel
var TaskALocs = Locales[..numTaskALocs];        L0 L1
var TaskBLocs = Locales[numTaskALocs+1..];     L2 L3 L4 L5 L6 L7

var CompGrid = Locales.reshape([1..gridRows, L0 L1 L2 L3
                          1..gridCols]);  L4 L5 L6 L7
```
Locale Methods

- The locale type supports built-in methods:
  ```python
  def locale.id: int; // index in LocaleSpace
  def locale.name: string; // similar to uname -n
  def locale.numCores: int; // # of processor cores
  def locale.physicalMemory(...): ...; // amount of memory
  ...
  ```

Executing on Remote Locales

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

- Semantics
  - Executes `stmt` on the locale specified by `expr`
  - Does not introduce concurrency

- Example
  ```python
  var A: [LocaleSpace] int;
  coforall loc in Locales do
    on loc { A(loc.id) = computation(loc.id); }
  ```
Querying a variable’s locale

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

- **Semantics**
  * Returns the locale on which `var-expr` is allocated

- **Example**

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

- **Output**

  ```
  0 1
  ```

Serial on-clause example

**on clauses**: indicate where code should execute

```plaintext
// Chapel programs begin running on locale 0 by default

var x, y: real; // allocate x & y on locale 0
on Locales(1) { // migrate task to locale 1
  var z: real; // allocate z on locale 1
  writeln(x.locale.id); // prints "0"
  writeln(z.locale.id); // prints "1"
  z = x + y; // requires "get" for x and y
  on Locales(0) do // migrate back to locale 0
    z = x + y; // requires "get" for z
    // return to locale 1
    // return to locale 0
```
Serial on-clause example (data-driven)

**on clauses**: indicate where code should execute

```
// Chapel programs begin running on locale 0 by default
var x, y: real;           // allocate x & y on locale 0
on Locales(1) {           // migrate task to locale 1
  var z: real;            // allocate z on locale 1
  writeln(x.locale.id);   // prints "0"
  writeln(z.locale.id);   // prints "1"
  z = x + y;              // optionally, in a data-driven manner
  on Locales(0) x do     // migrate back to locale 0
    z = x + y;           // requires "get" for z
  }                       // return to locale 1
                          // return to locale 0
```

Parallel on-clause examples

**on clauses**: indicate where code should execute

By naming locales explicitly...

```
cobegin {
  on TaskALocs do computeTaskA(...);  // return to locale 0
  on TaskBLocs do computeTaskB(...);  // return to locale 0
  on Locales[0] do computeTaskC(...);  // return to locale 0
}
```

...or in a data-driven manner...

```
computePivot(data, lo, hi);
cobegin {
  on data[lo] do Quicksort(data, lo, pivot);
  on data[pivot] do Quicksort(data, pivot, hi);
}
```
Here

- Built-in locale function
  ```chapel
def here: locale;
```

- Semantics
  - Returns the locale on which the task is executing

- Example
  ```chapel
  writeln(here.id);
  on Locales(1) do
    writeln(here.id);
  
  Output
  0
  1
  ```

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
  ```plaintext
  (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 are no remote references in `stmt`
  - Checked at runtime by default; can be disabled for performance

- Example

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

- Basics of Multi-Locale Chapel
- Domain and Array Distributions
  - overview
  - a case study: Block1D
- Sample Uses of Distributed Domains/Arrays

Chapel Distributions

*Distributions*: “Recipes for parallel, distributed arrays”
- help the compiler map from the computation’s global view…

…down to the *fragmented*, per-processor implementation
Domain Distribution

Domains may be distributed across locales

```chapel
var D: domain(2) distributed Block on CompGrid = ...;
```

A distribution implies...
...ownership of the domain’s indices (and its arrays’ elements)
...the default work ownership for operations on the domains/arrays
  - e.g., forall loops or promoted operations over domains/arrays

Authoring Distributions

- (Advanced) Programmers can write distributions in Chapel
- Chapel will support a standard library of distributions
  - *research goal:* using the same mechanism that users would
  - our compiler should have no knowledge of specific distributions
    - only its structural interface—how to...
      - create domains and arrays using that distribution
      - map indices to locales
      - access array elements
      - iterate over indices/array elements
        - sequentially
        - in parallel
        - in parallel and zippered with other parallel iteratable types
      ...and so forth...
- Distributions are built using the concepts we’ve already seen
  - on clauses for expressing what data & tasks each locale owns
  - begins, cobegins, coforalls to express inter- & intra-locale parallelism
Distributions

- All the domain types we’ve seen will support distributions
- Domain/array semantics are independent of distribution
  - performance and parallelism may vary greatly as distributions change
A Simple Distribution: Block1D

- **Goal**: block a 1D index space across a set of locales

Use a bounding box to compute the blocking

Distributions vs. Domains

**Q1**: Why distinguish between distributions and domains?
**Q2**: Why do distributions map an index *space* rather than a fixed index set?

**A**: To permit several domains to share a single distribution
  - amortizes the overheads of storing a distribution
  - supports trivial domain/array alignment and compiler optimizations

```c
const D : distributed B1 = [1..8], L0 L1 L2
outerD : distributed B1 = [0..9],
innerD : subdomain(D) = [2..7],
slideD : subdomain(D) = [4..6];
```

Sharing a distribution supports trivial alignment of these domains
Distributions vs. Domains

Q1: Why distinguish between distributions and domains?
Q2: Why do distributions map an index space rather than a fixed index set?

A: To permit several domains to share a single distribution
   • amortizes the overheads of storing a distribution
   • supports trivial domain/array alignment and compiler optimizations

```
const D : distributed B1 = [1..8],
outerD: distributed B2 = [0..9],
innerD: distributed B3 = [2..7],
slideD: distributed B4 = [4..6];
```

When each domain is given its own distribution, the alignment between indices is less obvious.

Chapel’s Distribution Architecture

<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><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>
</tbody>
</table>
Chapel's Distribution Architecture

**Global Descriptors**
- One global instance or replicated per locale
  - target locale set
  - distribution params
  - map index to locale
- Global index info
- Global value iteration
- Random access

**Local Descriptors**
- One instance per locale
  - local indices
  - local index iteration
  - Add new indices
- Local values
- Local value iteration
- Local random access

Legend:
- = descriptor state
- = descriptor methods

Block1D Distribution Classes

**Code**
- `const B1 = new Block1D(bbox=[1..8])`
- `const D: domain(1)`
- `distributed B1 = [1..8];`
- `var A: [D] real;`

**Global Descriptors**
- BoundingBox = [1..8]
- TargetLocales = L0 L1 L2

**Local Descriptors**
- L0 L1 L2
  - myChunk = [4..6]
  - myBlock = [4..5]
  - myElems = [4..5]
  - myChunk = [6..8]
  - myBlock = [6..8]
  - myElems = [6..8]
Block1D Distribution Classes

- **distribution**
  - `const B1 = new Block1D(bbox=[1..8])`
  - `(LocaleSpace = [0..2])`

- **domain**
  - `const sliceD: domain(1)`
  - `distributed B1 = [4..6]`

- **array**
  - `var A2: [sliceD] real;`

---

**Outline**

- Basics of Multi-Locale Chapel
- Domain and Array Distributions
- Sample Uses of Distributed Domains/Arrays
  - HPCC Stream Triad
  - HPCC Random Access (RA)
Introduction to STREAM Triad

Given: \( m \)-element vectors \( A, B, C \)

Compute: \( \forall i \in 1..m, A_i = B_i + \alpha \cdot C_i \)

Pictorially:

\[ \begin{align*}
A &= \begin{array}{c}
\text{bar graph} \\
\end{array} \\
B &= \begin{array}{c}
\text{bar graph} \\
\end{array} \\
C &= \begin{array}{c}
\text{bar graph} \\
\end{array}
\]

\[ = \]

\[ + \]

\[ \alpha \]

\[ A \quad B \quad C \quad \alpha \]

Introduction to STREAM Triad

Given: \( m \)-element vectors \( A, B, C \)

Compute: \( \forall i \in 1..m, A_i = B_i + \alpha \cdot C_i \)

Pictorially (in parallel):

\[ \begin{align*}
A &= \begin{array}{c}
\text{bar graph} \\
\end{array} \\
B &= \begin{array}{c}
\text{bar graph} \\
\end{array} \\
C &= \begin{array}{c}
\text{bar graph} \\
\end{array}
\]

\[ = \]

\[ + \]

\[ \alpha \]

\[ A \quad B \quad C \quad \alpha \]
STREAM Triad in Chapel

```chapel
const BlockDist = new Block1D(bbox=[1..m], tasksPerLocale=...);
```

```chapel
const ProblemSpace: domain(1, int(64)) distributed BlockDist = [1..m];
```

```chapel
var A, B, C: [ProblemSpace] real;
```

```chapel
forall (a, b, c) in (A, B, C) do
    a = b + alpha * c;
```

Introduction to Random Access

**Given:** $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)

**Compute:** $N_U$ random updates to the table using bitwise-xor

**Pictorially:**

```plaintext
[Diagram showing random updates]
```
Introduction to Random Access

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))

Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & 8 & 9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 & 17 & 18 \\
19 & 20 & 21 & 22 & 23 & 24 \\
25 & 26 & 27 & 28 & 29 & 30 \\
31 & 32 & 33 & 34 & 35 & 36 \\
37 & 38 & 39 & 40 & 41 & 42 \\
43 & 44 & 45 & 46 & 47 & 48 \\
49 & 50 & 51 & 52 & 53 & 54 \\
55 & 56 & 57 & 58 & 59 & 60 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 2 & 1 & 3 & 4 & 5 \\
6 & 7 & 8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 & 16 & 17 \\
18 & 19 & 20 & 21 & 22 & 23 \\
24 & 25 & 26 & 27 & 28 & 29 \\
30 & 31 & 32 & 33 & 34 & 35 \\
36 & 37 & 38 & 39 & 40 & 41 \\
42 & 43 & 44 & 45 & 46 & 47 \\
48 & 49 & 50 & 51 & 52 & 53 \\
54 & 55 & 56 & 57 & 58 & 59 \\
\end{array}
\]

\[
0 \oplus 21 \Rightarrow \text{the value } 21 \text{ into } T_{(21 \mod m)}
\]

Repeat \( N_U \) times
Introduction to Random Access

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))

Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially (in parallel):

Random Numbers
Not actually generated using lotto ping-pong balls!
Instead, implement a pseudo-random stream:
- \( k \)th random value can be generated at some cost
- given the \( k \)th random value, can generate the \((k+1)\)-st much more cheaply
Introduction to Random Access

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))

Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially (in parallel):

Conflicts
When a conflict occurs an update may be lost; a certain number of these are permitted

RA Declarations in Chapel

```chapel
const TableDist = new Block1D(bbox=[0..m-1], tasksPerLocale=...),
UpdateDist = new Block1D(bbox=[0..N_U-1], tasksPerLocale=...);

const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);
```
RA Computation in Chapel

const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
   Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

forall (_, r) in (Updates, RAStream()) do
  on T(r&indexMask) do
    T(r&indexMask) ^= r;
RA Computation in Chapel: tune for affinity

```chapel
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1], Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

forall (_, r) in (Updates, RAStream()) do
  on T(r&indexMask) do
    T(r&indexMask) ^= r;
```

### Diagram
- **RAStream()**: r_0, r_1, r_2, ..., r_{N_U-1}
- **T**: T_0, T_1, T_2, ..., T_{N_U-1}

 Chung-Ling Hu, University of Texas at Austin

Chapel: Locality and Affinity (43)

RA Computation in Chapel: fire and forget

```chapel
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1], Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

sync {
  forall (_, r) in (Updates, RAStream()) do
    on T(r&indexMask) do
      begin T(r&indexMask) ^= r;
```

### Diagram
- **RAStream()**: r_0, r_1, r_2, ..., r_{N_U-1}
- **T**: T_0, T_1, T_2, ..., T_{N_U-1}

 Chung-Ling Hu, University of Texas at Austin

Chapel: Locality and Affinity (44)

22Cray Proprietary
Locality and Affinity Status

- **Stable Features:**
  - locale types, methods, and variables
  - on clauses

- **Incomplete Features:**
  - the local block has not been stress-tested
  - we’ve only just started getting our first distributions working
    - this is the reason that foralls/promotions don’t result in parallelism
    - only Block1D and only for basic domain/array operations
      - see examples/hpcc/stream.chpl and ra.chpl for sample uses

- **Future Directions:**
  - improved support for replicated, symmetric data
  - distributions as a mechanism for software resiliency
  - richer locale types: multiple flavors and hierarchical locales
    - to better represent machine structure and heterogeneity

Questions?