Chapel’s Downward-Facing Interfaces

Brad Chamberlain
Cray Inc.
March 2011
Goal Of This Talk

• Approach the topic of mapping Chapel to a new target platform by...
  ...reviewing some core Chapel concepts
  ...describing how Chapel’s downward-facing interfaces implement those concepts
Disclaimers

- All Chapel interfaces are subject to continued evolution based on ongoing experience & improvements
  (e.g., if mapping to a new technology requires changes, we’re open to that)
Compiling Chapel

Chapel Source Code → chpl → Chapel Standard Modules → Chapel Executable
Chapel Compiler Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Chapel Standard Modules
- Internal Modules (written in Chapel)
- Runtime Support Libraries (in C)
  - Tasks/Threads
  - Communication
  - Memory
Chapel Runtime Library Architecture

Chapel Runtime Support Libraries (written in C)

- Tasks/Threads
- Communication
- Memory
- Timers
- Launchers
- Standard

Standard and third-party libraries
Tasking/Threading Interface

- **Role:** Responsible for parallelism/synchronization
- **Main Focus:**
  - support begin/cobegin/coforall statements
  - support synchronization variables
Begin Statements

- **Syntax**
  \[
  \text{begin-stmt}: \begin{array}{l}
  \text{begin stmt}
  \end{array}
  \]

- **Semantics**
  - Creates a task to execute \textit{stmt}
  - Original ("parent") task continues without waiting

- **Example**
  \[
  \begin{array}{l}
  \text{begin writeln("hello world");}
  \text{writeln("good bye");}
  \end{array}
  \]

- **Possible output**
  \[
  \begin{array}{l}
  \text{hello world}
  \text{good bye}
  \end{array}
  \begin{array}{l}
  \text{good bye}
  \text{hello world}
  \end{array}
  \]
Block-Structured Task Creation: Cobegin

• **Syntax**

```plaintext
cobegin-stmt:
  cobegin { stmt-list }
```

• **Semantics**
  • Creates a task for each statement in `stmt-list`
  • Parent task waits for all sub-tasks to complete

• **Example**

```plaintext
cobegin {
  consumer(1);
  consumer(2);
  producer();
}  // wait here for all three tasks to terminate
```
Loop-Structured Task Invocation: Coforall

- **Syntax**
  
  ```
  coforall-loop:
  coforall index-expr in iterable-expr { stmt-list }
  ```

- **Semantics**
  - Create a task for each iteration in `iteratable-expr`
  - Parent task waits for all sub-tasks to complete

- **Example**

  ```
  begin producer();
  coforall i in 1..numConsumers {
    consumer(i);
  } // wait here for all consumers to terminate
  ```
Synchronization Variables

• **Syntax**

\[
\text{sync-type:} \quad \text{sync type}\]

• **Semantics**

  - Stores *full/empty* state along with normal value
  - Default read blocks until *full*, leaves *empty*
  - Default write blocks until *empty*, leaves *full*
  - Other variations supported via method calls (e.g., .readFF())

• **Examples: Critical sections and futures**

```plaintext
var future$: sync real;

begin future$ = compute();
computeSomethingElse();
useComputedResults(future$);
```
var buff$: [0..#buffersize] sync real;

cobegin {
    producer();
    consumer();
}

proc producer() {
    var i = 0;
    for ... {
        i = (i+1) % buffersize;
        buff$(i) = ...;
    }
}

proc consumer() {
    var i = 0;
    while ... {
        i= (i+1) % buffersize;
        ...buff$(i)...;
    }
}
Runtime Tasking Interface

- **Startup/Teardown**
- **Singleton Tasks** (to implement `begin`):
  - fire and forget
- **Task Lists** (to implement `cobegin/coforall`):
  - create, execute, free
- **Synchronization** (to implement sync variables):
  - lock/unlock, wait full/empty, mark full/empty, isfull
- **Control**:
  - yield/sleep
- **Queries**:
  - #tasks running/queued/blocked, task state, ...
• **Distinguish *may* vs. *must* tasks**
  - e.g., binary tree search “may” use multiple tasks; producer/consumer “must”
  - today all Chapel tasks are *must*
    ⇒ always correct, but not as amenable to runtime throttling techniques

• **Task-Private Variables**

• **Task Teams**

• **Tasking Policies**
  - e.g., “Can this task be work-stolen locally/remotely?”

• **Task Prioritization(?)**
Chapel Runtime Support Libraries (written in C)

- FIFO Task Pool
- Cray XMT HW Tasks
- Nanos++ Tasks (BSC)
- Qthreads Tasks (Sandia)
- Thread Pool
- POSIX Threads

Tasking:
- Threading (optionally)

Synchronization:
- Nanos++
- Qthreads

Runtime Tasking Interface: Instantiations
Chapel Runtime Library Architecture

Chapel Runtime Support Libraries (written in C)

- Tasks/Threads
- Communication
- Memory
- Timers
- Launchers
- Standard

Standard and third-party libraries
Communication Interface

- **Role:** Responsible for inter-node communication
- **Main Focus:**
  - establishment of locales
  - active messages
  - single-sided puts/gets
Chapel's *Locale* Type

- **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
The On Statement

• Syntax

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

• Semantics

  • Executes `stmt` on the locale that stores `expr`

• Example

``` writeln("start on locale 0");
on Locales[1] do // uses an active message
  writeln("now on locale 1");
writeln("on locale 0 again"); ```
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 gets of x and y
}

on Locales[0] do  // migrate task to locale 0
  z = x + y;      // remote put to z
  // migrate back to locale 1

on x do            // data-driven migration to locale 0
  z = x + y;      // remote put to z
  // migrate back to locale 1
  // migrate back to locale 0

Locale 0  | x  
| y  
Locale 1  | z  

Runtime Communication Interface

- **Startup/Teardown**
  - including establishment of locales, memory registration, setup of global variables/consts, global barriers, option to run in gdb, termination

- **Single-Sided Communication:**
  - put/get blocks of data

- **Active Messages:**
  - blocking/nonblocking fork

- **Diagnostics:**
  - trace/count communication events

- **Optional Task-layer Hooks:**
  - e.g., ability to switch tasks on communication events
Runtime Comm. Interface: Future Directions

- **Richer Styles of Puts/Gets**
  - strided, scatter/gather, etc.
- **Collectives**
  - implemented via puts/gets today
Chapel Runtime Support Libraries (written in C)
Chapel Runtime Library Architecture

Chapel Runtime Support Libraries (written in C)

Tasks/Threads
Communication
Memory
Timers
Launchers
Standard

Standard and third-party libraries
Other Runtime Interfaces

- **Memory:**
  - malloc/realloc/free

- **Timers:**
  - query time

- **Launchers:**
  - queue and/or launch binaries

- **Standard:**
  - argument parsing, I/O, type conversions, system queries, memory tracking, ...
Other Runtime Interfaces: Instantiations

Chapel Runtime Support Libraries (written in C)

- Memory
- Timers
- Launchers
- Standard

C runtime
- dmalloc
- C runtime
- Cray XMT timers
- GASNet scripts
- aprun
- C runtime

- PBS
- loadleveler
- LSF
- mtarun
- PVM
- mpirun
- SLURM
- ...
Q: “This all seems fairly low-level... What about all those sweet productivity features?”

A1: Many are built into the compiler
   - type inference
   - OOP
   - iterators

A2: For others, recall Chapel’s multiresolution design:
Global STREAM Triad in Chapel

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

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

A = B + alpha * C;
```
Domain Maps in Chapel

**Domain Maps**: “recipes for parallel/distributed arrays” (and index sets)

Domain maps define:
- Ownership of domain indices and array elements
- Underlying representation of indices and elements
- Standard operations on domains and arrays
  - E.g., iteration, slicing, access, reindexing, rank change

Domain maps are built using Chapel concepts
- classes, iterators, type inference, generic types
- task parallelism
- locales and on-clauses
- other domains and arrays
Global STREAM Triad in Chapel

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

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

A = B + alpha * C;
```

This domain’s declaration did not specify a domain map, so it gets the compiler-provided default. In practice this typically maps the domain indices/array elements to the current locale and uses the locally available parallelism to execute forall loops.
const ProblemSpace: domain(1, int(64))

    dmapped Block(boundingBox=[1..m])
    = [1..m];

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

A = B + alpha * C;
const ProblemSpace: domain(1, int(64))
    dmapped Cyclic(startIdx=1)
    = [1..m];

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

A = B + alpha * C;
Promoted scalar operators/function calls...

\[ A = B + \alpha \cdot C; \]

...are defined in terms of zippered data parallel forall loops:

```plaintext
forall (a,b,c) in (A,B,C) do
    a = b + \alpha \cdot c;
```
Zippered data parallel forall loops:

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

...are defined in terms of leader/follower iterators:

- **leader iterator**: introduces parallelism, assigns work to tasks
- **follower iterators**: serially execute work assigned by leader
- in this example, array A is the leader; A, B, and C are all followers
- **conceptually**, the Chapel compiler generates something like:

```chapel
for work in A.lead () do
    for (a, b, c) in (A.follow(work), B.follow(work),
                     C.follow(work)) do
        a = b + alpha * c;
```

Leader iterators are defined in terms of task/locality features:

```plaintext
iter BlockArr.lead() {
    coforall loc in Locales do
        on loc do
            coforall tid in here.numCores do
                yield computeMyWork(loc.id, tid);
}
```

Follower iterators simply use serial base language features:

```plaintext
iter BlockArr.follow(work) {
    for i in work do
        yield accessElement(i);
}
```
Similarly, storage for distributed arrays uses locality and base language features. Here’s a schematic of what we want:

**Locale 0**

- **A:** `BlockArr(real)`
  - `localBlocks`
  - `LocalBlockArr(real)`
    - `data`

**Locale 1**

- `LocalBlockArr(real)`
  - `data`

**Locale 2**

- `LocalBlockArr(real)`
  - `data`

**Locale 3**

- `LocalBlockArr(real)`
  - `data`
Similarly, storage for distributed arrays uses locality and base language features. Here’s a sketch in code:

```plaintext
class LocalBlockArr {
    type eltType;  // generic field for array element type
    var data: [...] eltType;  // a non-distributed array of values
}

class BlockArr {
    type eltType;
    // local array of (potentially remote) class references
    var localBlocks: [targetLocaleSpace] LocalBlockArr(eltType);

def BlockArr() {
    // allocate a LocalBlockArr instance per locale on that locale
    coforall loc in targetLocaleSpace do
        on targetLocales[loc] do
            localBlocks[loc] = new LocalBlockArr(eltType);
}
```
Q: “So where does all this code specifying distributed data structures and higher-level data parallelism reside?”

A1: In the Chapel modules
Chapel Compiler Architecture

Chapel Compiler

Chapel Source Code

Chapel-to-C Compiler

Generated C Code

Standard C Compiler & Linker

Chapel Executable

Chapel Standard Modules

Internal Modules (written in Chapel)

Runtime Support Libraries (in C)

Tasks/Threads

Communication

Memory

...
Q: “So where does all this code specifying distributed data structures and higher-level data parallelism reside?”

A1: In the Chapel modules
- Internal modules define default domain maps (layouts) for non-distributed domains and arrays
- Standard modules define a library of other domain maps (both layouts and distributions)

A2: In user code (users can write domain maps using the same techniques that we do)
Domain Maps fall into two major categories:

**layouts**: target a single locale (shared memory)
- e.g., a desktop machine or multicore node
- **examples**: row- and column-major order, tilings, compressed sparse row

**distributions**: target distinct locales (distributed mem.)
- e.g., a distributed memory cluster or supercomputer
- **examples**: Block, Cyclic, Block-Cyclic, Recursive Bisection, ...
**Domain Map**

- **Represents:** a domain map value
- **Generic w.r.t.:** index type
- **State:** domain map representation
- **Size:** $\Theta(1)$
- **Required Interface:**
  - create new domains
- **Other Interfaces:** ...

**Domain**

- **Represents:** a domain value
- **Generic w.r.t.:** index type
- **State:** representation of index set
- **Size:** $\Theta(1) \rightarrow \Theta(numIndices)$
- **Required Interface:**
  - create new arrays
  - query size and membership
  - serial, parallel, zippered iteration
  - domain assignment
  - intersections and orderings
  - add, remove, clear indices
- **Other Interfaces:** ...

**Array**

- **Represents:** an array
- **Generic w.r.t.:** index type, element type
- **State:** array elements
- **Size:** $\Theta(numIndices)$
- **Required Interface:**
  - (re-)allocation of array data
  - random access
  - serial, parallel, zippered iteration
  - slicing, reindexing, rank change
  - get/set of sparse “zero” values
- **Other Interfaces:** ...

---

**Descriptors for Layouts**

- **Domain Map**
  - Represents: a domain map value
  - Generic w.r.t.: index type
  - State: domain map representation
  - Size: $\Theta(1)$
  - Required Interface: create new domains
  - Other Interfaces: ...

- **Domain**
  - Represents: a domain value
  - Generic w.r.t.: index type
  - State: representation of index set
  - Size: $\Theta(1) \rightarrow \Theta(numIndices)$
  - Required Interface: create new arrays, query size and membership, serial, parallel, zippered iteration, domain assignment, intersections and orderings, add, remove, clear indices
  - Other Interfaces: ...

- **Array**
  - Represents: an array
  - Generic w.r.t.: index type, element type
  - State: array elements
  - Size: $\Theta(numIndices)$
  - Required Interface: (re-)allocation of array data, random access, serial, parallel, zippered iteration, slicing, reindexing, rank change, get/set of sparse “zero” values
  - Other Interfaces: ...

---

**Cray The Supercomputer Company**
### Descriptors for Distributions

<table>
<thead>
<tr>
<th>Global</th>
<th>Domain Map</th>
<th>Domain</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>one instance per object (logically)</td>
<td><strong>Role:</strong> Similar to layout’s domain map descriptor</td>
<td><strong>Role:</strong> Similar to layout’s domain descriptor, but no $\Theta(#indices)$ storage</td>
<td><strong>Role:</strong> Similar to layout’s array descriptor, but data is moved to local descriptors</td>
</tr>
<tr>
<td></td>
<td><strong>Size:</strong> $\Theta(1) \rightarrow \Theta(#locales)$</td>
<td></td>
<td><strong>Size:</strong> $\Theta(1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local</th>
<th>Domain Map</th>
<th>Domain</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>one instance per locale per object (typically)</td>
<td><strong>Role:</strong> Stores node-specific domain map parameters</td>
<td><strong>Role:</strong> Stores node’s subset of domain’s index set</td>
<td><strong>Role:</strong> Stores node’s subset of array’s elements</td>
</tr>
<tr>
<td></td>
<td><strong>Size:</strong> $\Theta(1) \rightarrow \Theta(#indices/#locales)$</td>
<td></td>
<td><strong>Size:</strong> $\Theta(#indices/#locales)$</td>
</tr>
</tbody>
</table>
Chapel supports an (ever-improving) extern interface that permits C types, variables, and functions to be prototyped and used within Chapel code. This can be a good way to prototype new functionality without changes to the compiler and runtime.

Domain maps in my slides are fairly static/simple; in practice they can be much more dynamic. i.e., nothing in the leader iterator’s interface prevents it from dynamically assigning work to tasks, creating/destroying tasks, migrating work, etc.

In HPCS, resiliency was owned by the HW/OS; at exascale, would be nice to have more resiliency concepts in Chapel itself.
Summary

- Mapping Chapel to a new architecture tends to require mapping tasking & communication layers
  - other stuff is portable or built on top of these
  - hierarchical locale concept is the tricky bit for exascale
1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly

2. Advanced users can write their own domain maps in Chapel
   - to cope with shortcomings in our standard library

3. Chapel’s standard layouts and distributions are written using the same user-defined domain map framework
   - to keep us honest and avoid falling over a performance cliff when moving from “built-in” to user-defined domain maps

4. Domain maps should typically only affect implementation and performance, not semantics
   - to support switching between domain maps effortlessly

One possible interpretation of Chapel’s design: What would you want in a language to support user-defined distributions well?
Why Chapel != 10·HPF, IMO (or even 1·HPF)

• HPF said very little about how similar whole-array operations were defined
  • particularly in the event of absent/contradictory directives
  • required a lot of cleverness/evaluation from the compiler to decide how to implement them efficiently
  • led to portability problems between compilers

• By contrast, such operations are well-defined in Chapel
  • implementation amounts to mechanical rewritings
  • details defined externally to the compiler via domain maps
    • written in Chapel, whether user-defined or standard
    • ⇒ semantics as portable as the domain maps themselves
A: Chapel has had the chance to learn from HPF’s mistakes (and other languages’ successes and failures)

- Why did HPF fail?
  - lack of sufficient performance soon enough
  - vagueness in execution/implementation model
  - only supported a single level of data parallelism, no task/nested
  - inability to drop to lower levels of parallel programming
  - lack of rich data parallel abstractions
  - fixed set of limited distributions on dense arrays
  - lack of an open source implementation
  - too based on Fortran for modern programmers
  - ...?

- The failure of one language, even a federally-backed and well-funded one, does not dictate the failure of all future languages
What is the role of the Chapel compiler?

- To implement the base language
- To implement task parallelism
- To know how to rewrite data parallelism
- To identify common communication patterns and rewrite to domain map interfaces that better handle them