Chapel's Downward-Facing Interfaces

Brad Chamberlain

Cray Inc.

March 2011

Goal Of This Talk

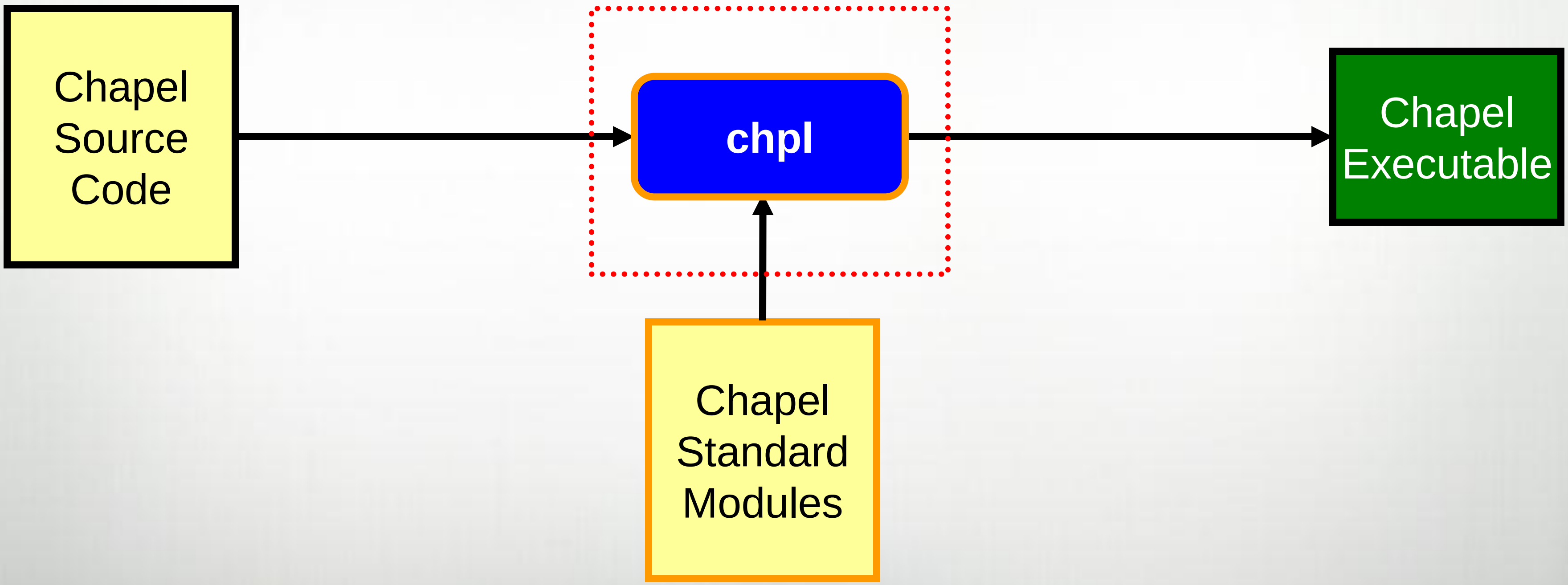
- Approach the topic of mapping Chapel to a new target platform by…
  - …reviewing some core Chapel concepts
  - …describing how Chapel's downward-facing interfaces implement those concepts

Disclaimers

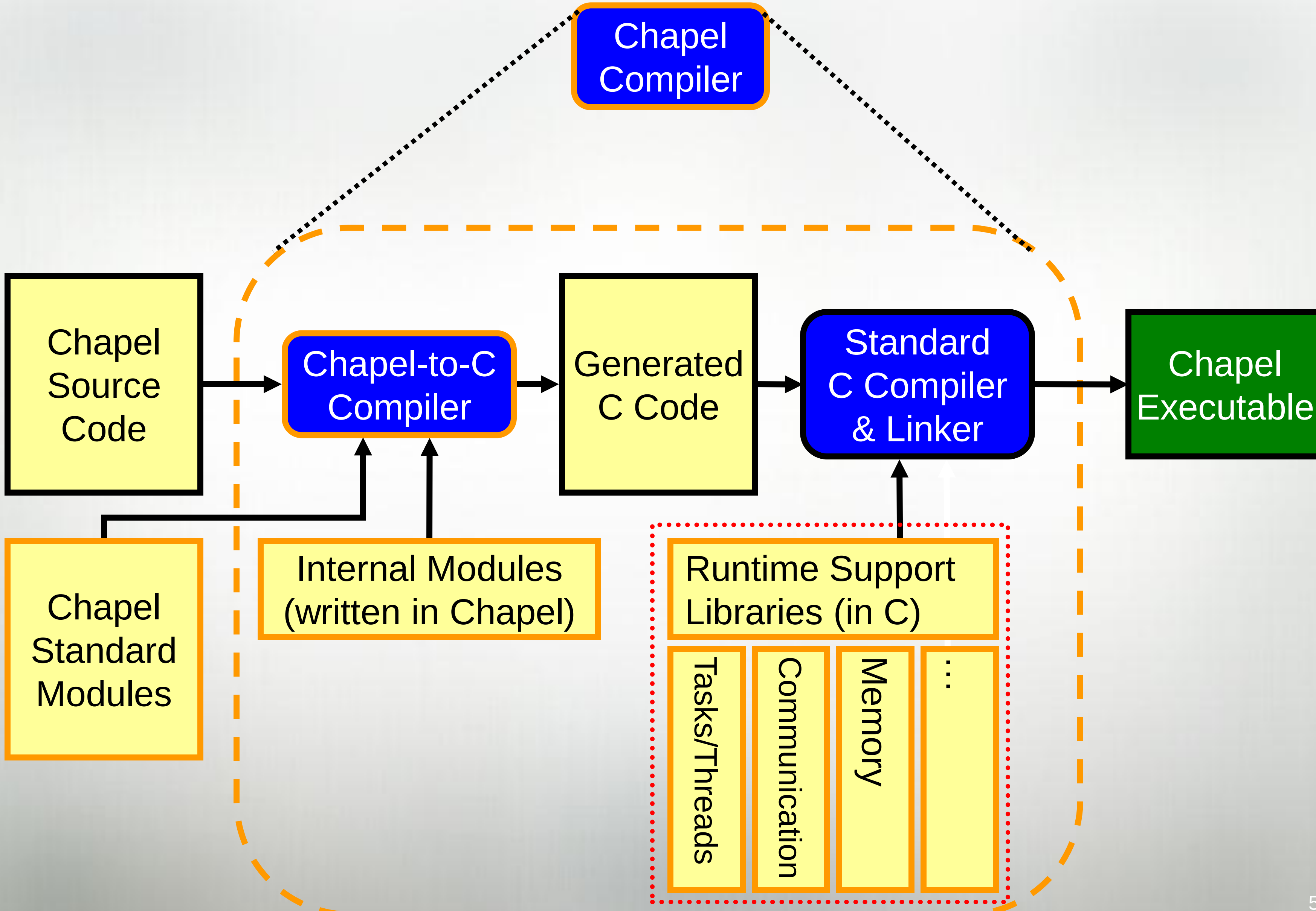
- All Chapel interfaces are subject to continued evolution based on ongoing experience & improvements

  (e.g., if mapping to a new technology requires changes, we're open to that)

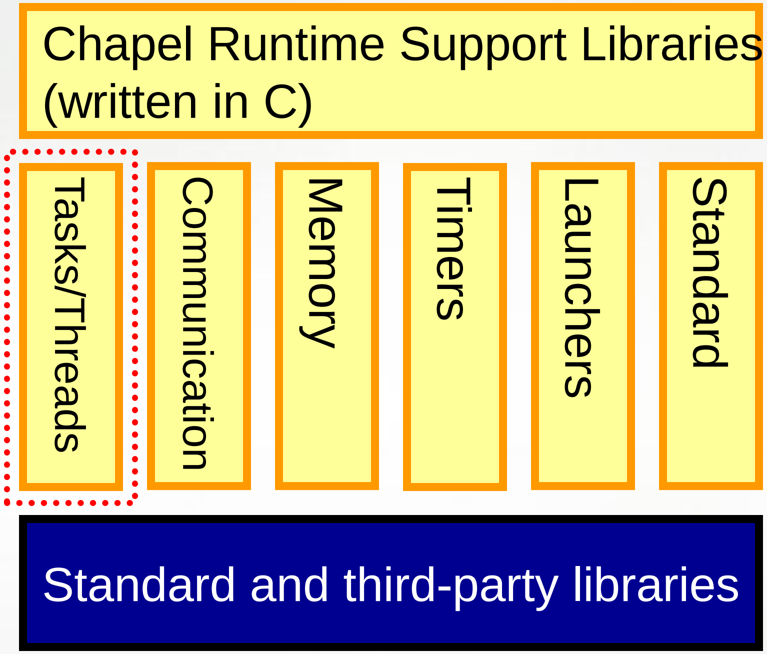
Compiling Chapel



Chapel Compiler Architecture



Chapel Runtime Library Architecture



Tasking/Threading Interface

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

Begin Statements

- Syntax

```
begin-stmt:
  begin stmt
```

- Semantics
  - Creates a task to execute *stmt*
  - Original (“parent”) task continues without waiting
- Example

```
begin writeln(“hello world”);
writeln(“good bye”);
```

- Possible output

```
hello world
good bye
```

```
good bye
hello world
```

Block-Structured Task Creation: Cobegin

- Syntax

```
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

```
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 iteratable-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

```
sync-type:
  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

```
var future$: sync real;

begin future$ = compute();
computeSomethingElse();
useComputedResults(future$);
```

Bounded Buffer Producer/Consumer Example

```
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)…;
  }
}
```

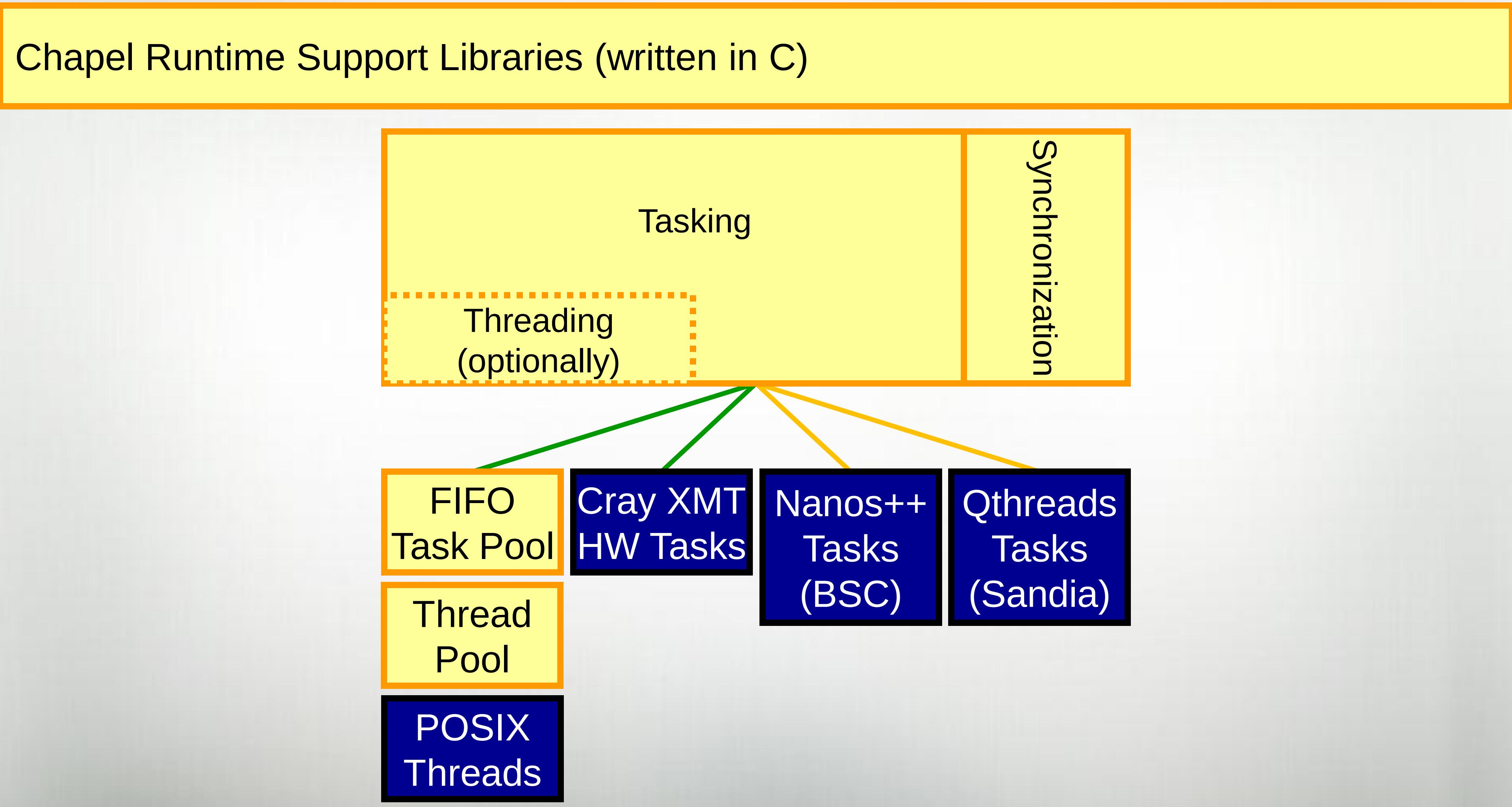
Runtime Tasking Interface

- **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, …

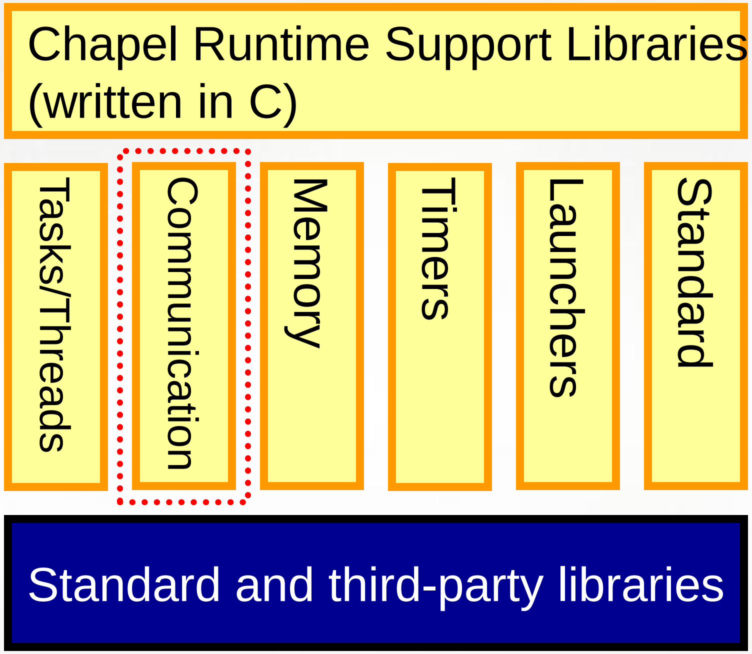
Runtime Tasking Interface: Future Directions

- **Distinguish *may* vs. *must* tasks**
  - e.g., binary tree search "may" use multiple tasks; producer/consumer "must"
  - today all Chapel tasks are *must*
    - $\Rightarrow$ 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(?)**

Runtime Tasking Interface: Instantiations



Chapel Runtime Library Architecture



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");
```

Serial Example with Implicit Communication

```
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 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*

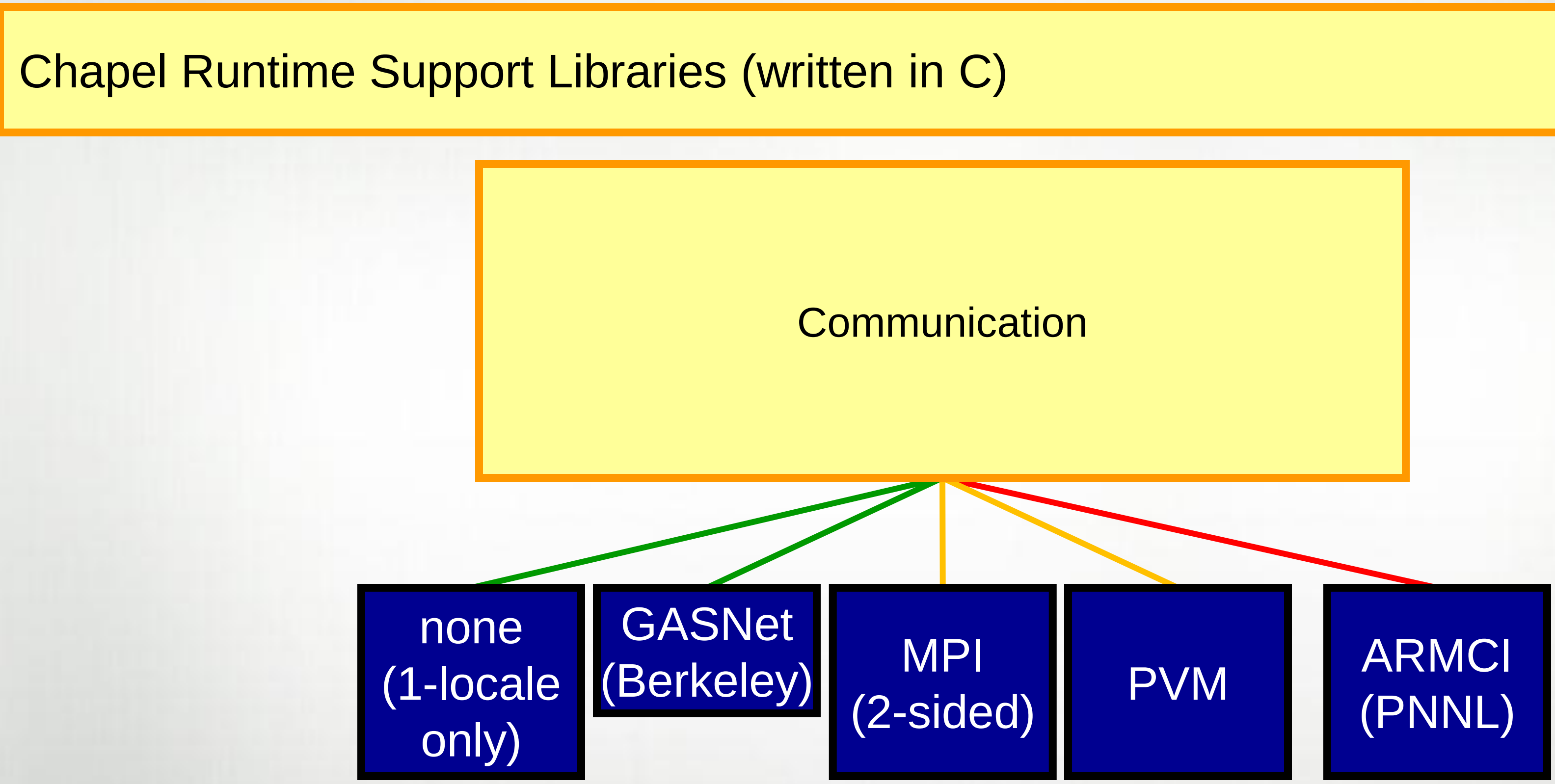
Runtime Communication Interface

- **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

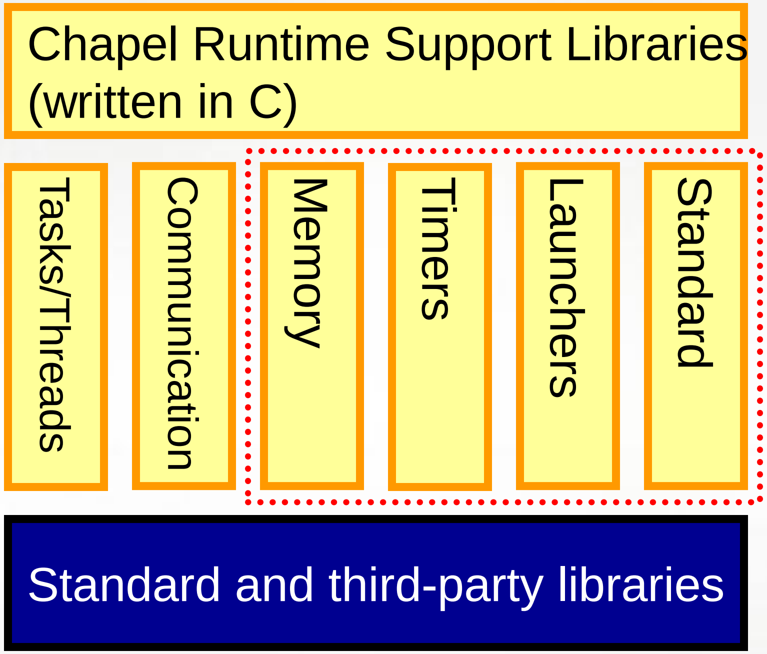
Runtime Comm. Interface: Future Directions

- **Richer Styles of Puts/Gets**
  - strided, scatter/gather, etc.
- **Collectives**
  - implemented via puts/gets today

Runtime Comm. Interface: Instantiations



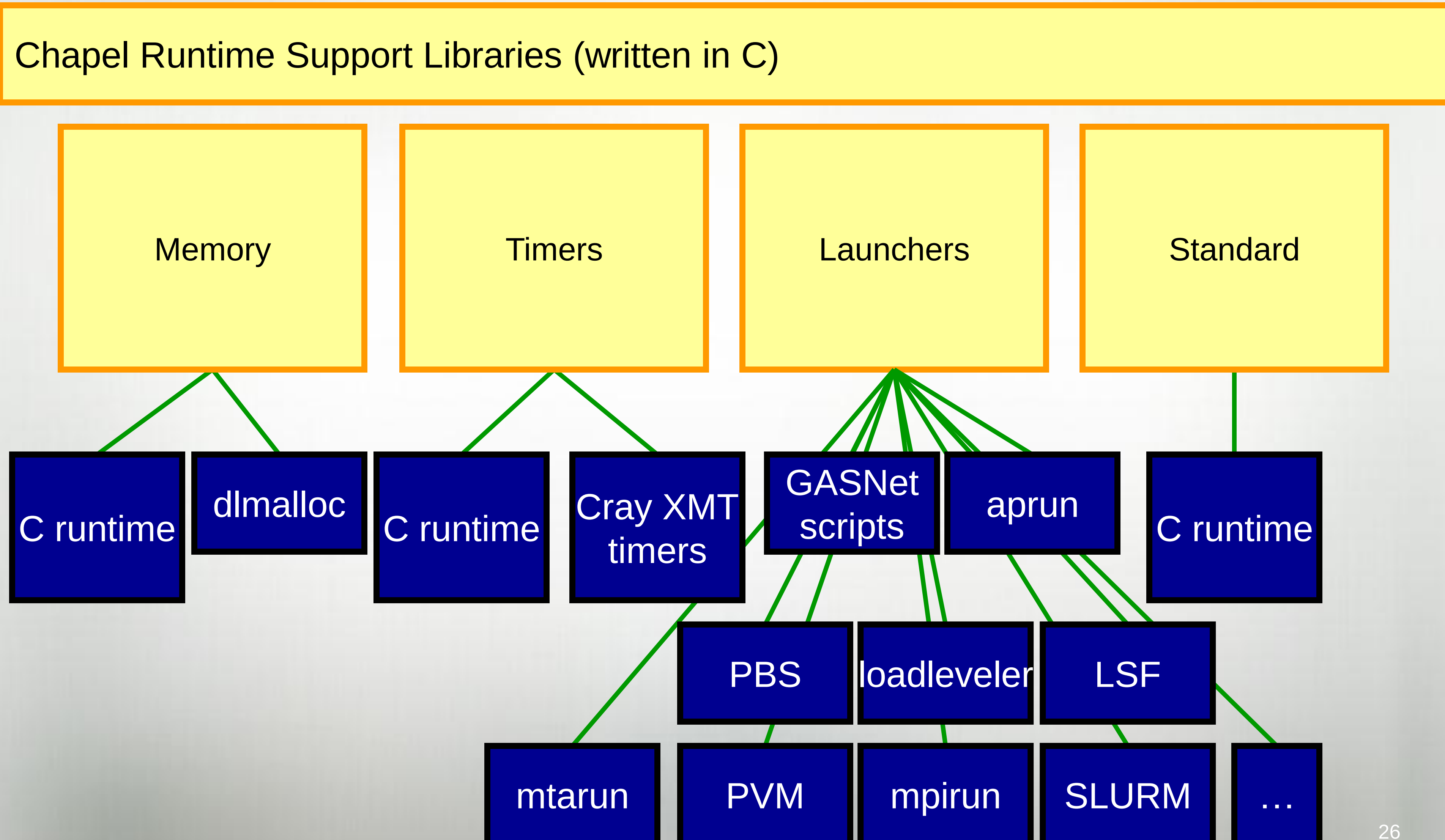
Chapel Runtime Library Architecture



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



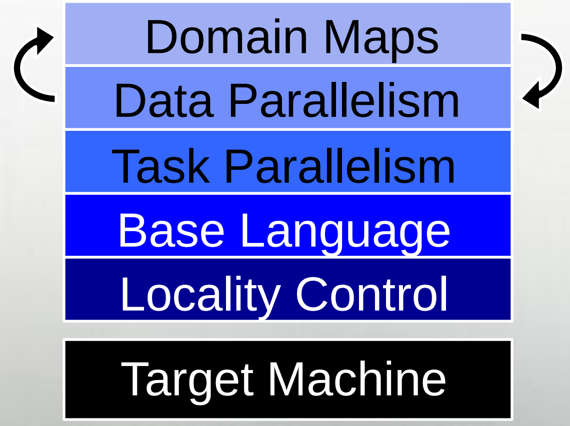
What Else?

Q: "This all seems fairly low-level… What about all those sweet productivity features?"

A1: Many are built into the compiler

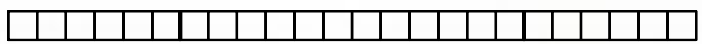
- type inference
- OOP
- iterators

A2: For others, recall Chapel's multiresolution design:

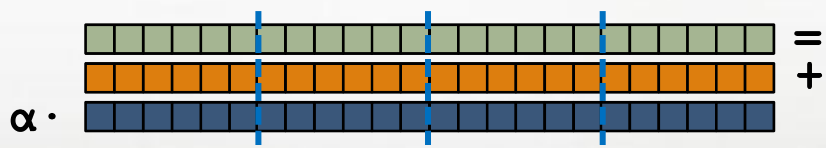


Global STREAM Triad in Chapel

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



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




```
A = B + alpha * C;
```

Domain Maps in Chapel

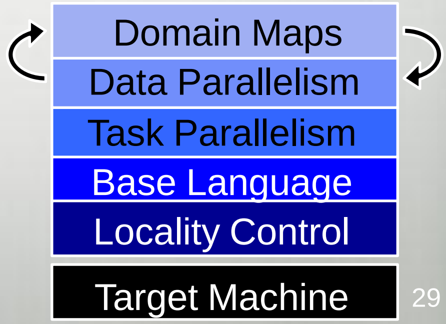
Domain Maps: "recipes for parallel/distributed arrays" (and index sets)

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

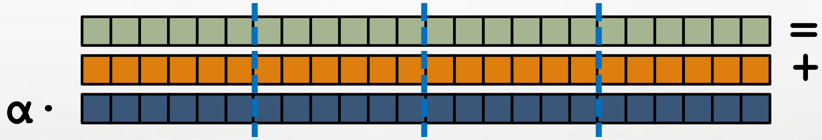


Global STREAM Triad in Chapel

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

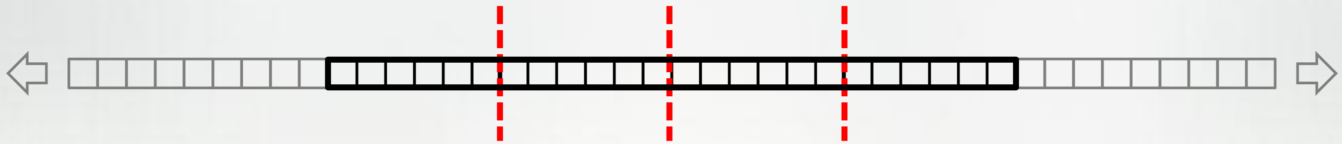
This domain's declaration did not specify a domain map, so it gets the compiler-provided default. In practice this typically maps the domain indices/array elements to the current locale and uses the locally available parallelism to execute forall loops

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

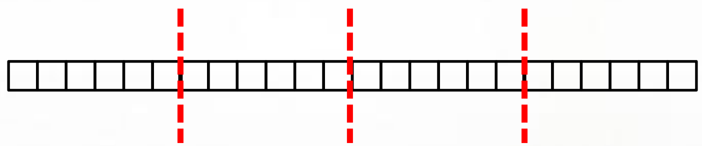



```
A = B + alpha * C;
```

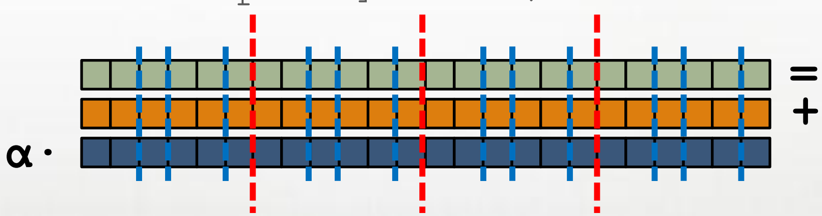
Global STREAM Triad in Chapel



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

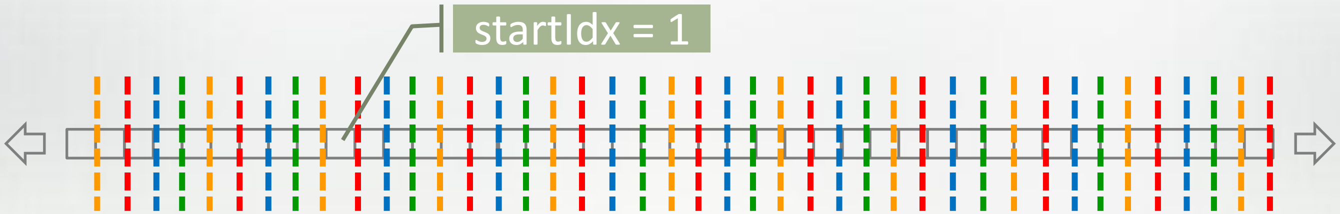


```
var A, B, C: [ProblemSpace] real;
```

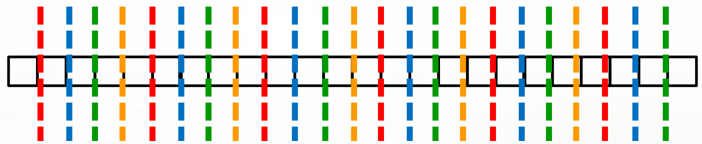



```
A = B + alpha * C;
```

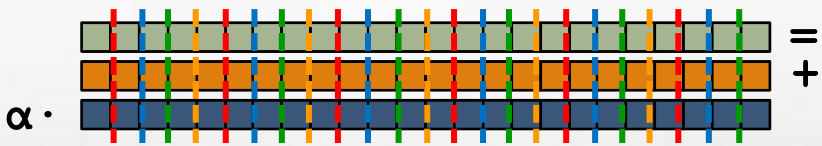
Global STREAM Triad in Chapel




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



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

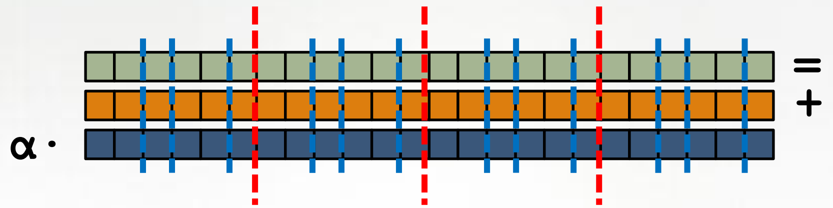



```
A = B + alpha * C;
```

Multiresolution Design: Layering the Features

Promoted scalar operators/function calls…

```
A = B + alpha*C;
```



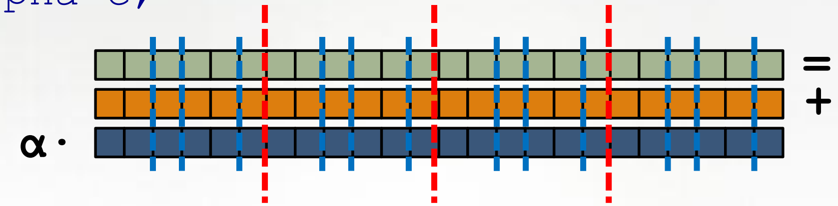
…are defined in terms of zippered data parallel forall loops:

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

Multiresolution Design: Layering the Features

Zippered data parallel forall loops:

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



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

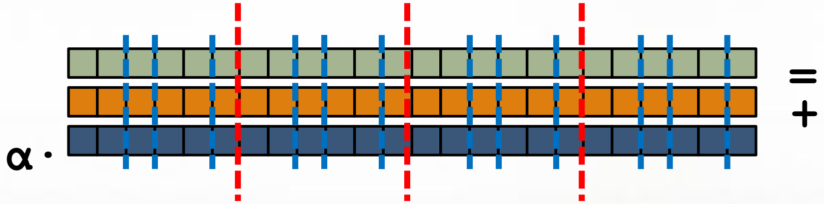
- *leader iterator:* introduces parallelism, assigns work to tasks
- *follower iterators:* serially execute work assigned by leader
- in this example, array A is the leader; A, B, and C are all followers
- *conceptually*, the Chapel compiler generates something like:

```
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;
```

Multiresolution Design: Layering the Features

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

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

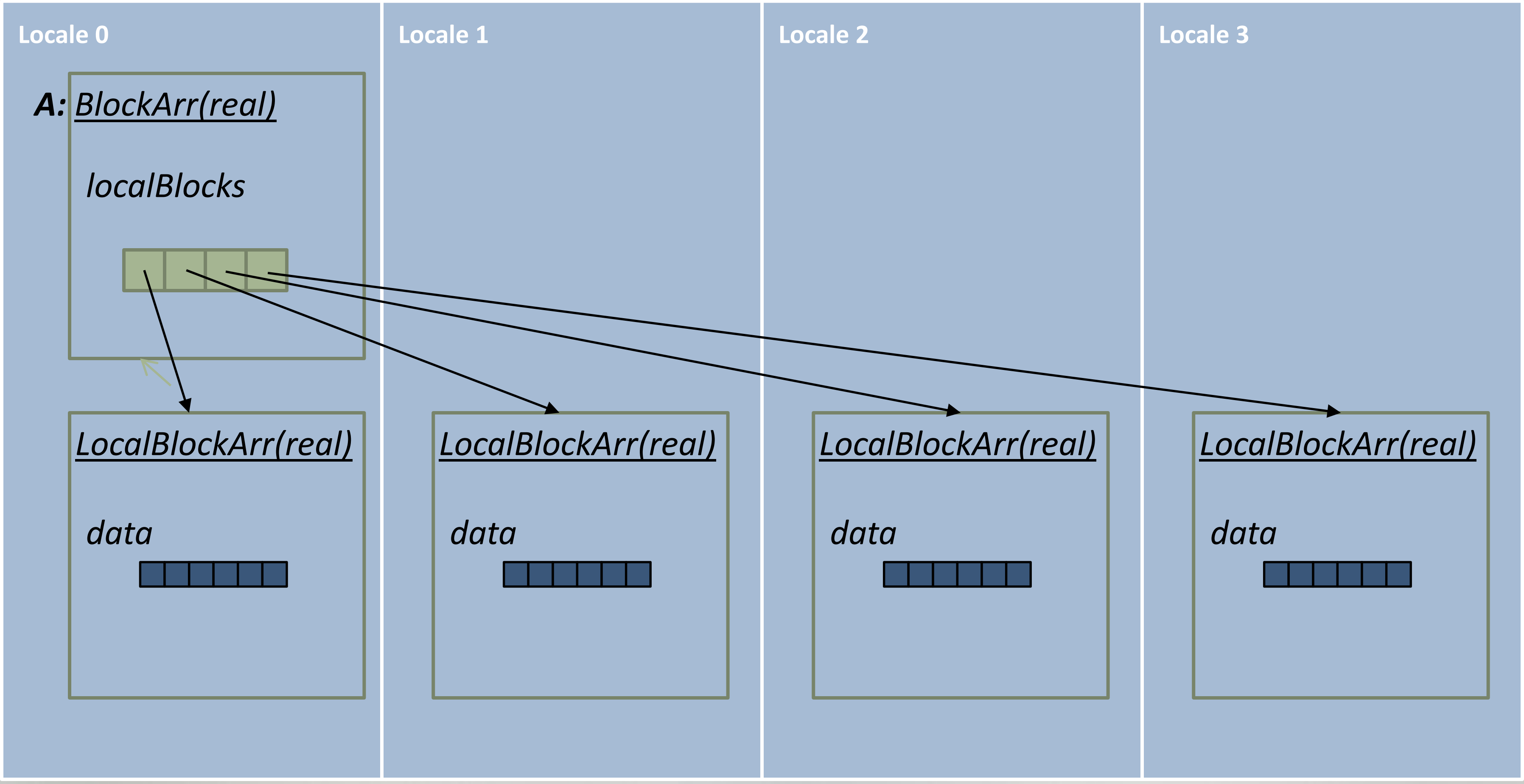


Follower iterators simply use serial base language features:

```
iter BlockArr.follow(work) {
  for i in work do
    yield accessElement(i);
}
```

Multiresolution Design: Layering the Features

Similarly, storage for distributed arrays uses locality and base language features. Here's a schematic of what we want:



Multiresolution Design: Layering the Features

Similarly, storage for distributed arrays uses locality and base language features. Here's a sketch in code:

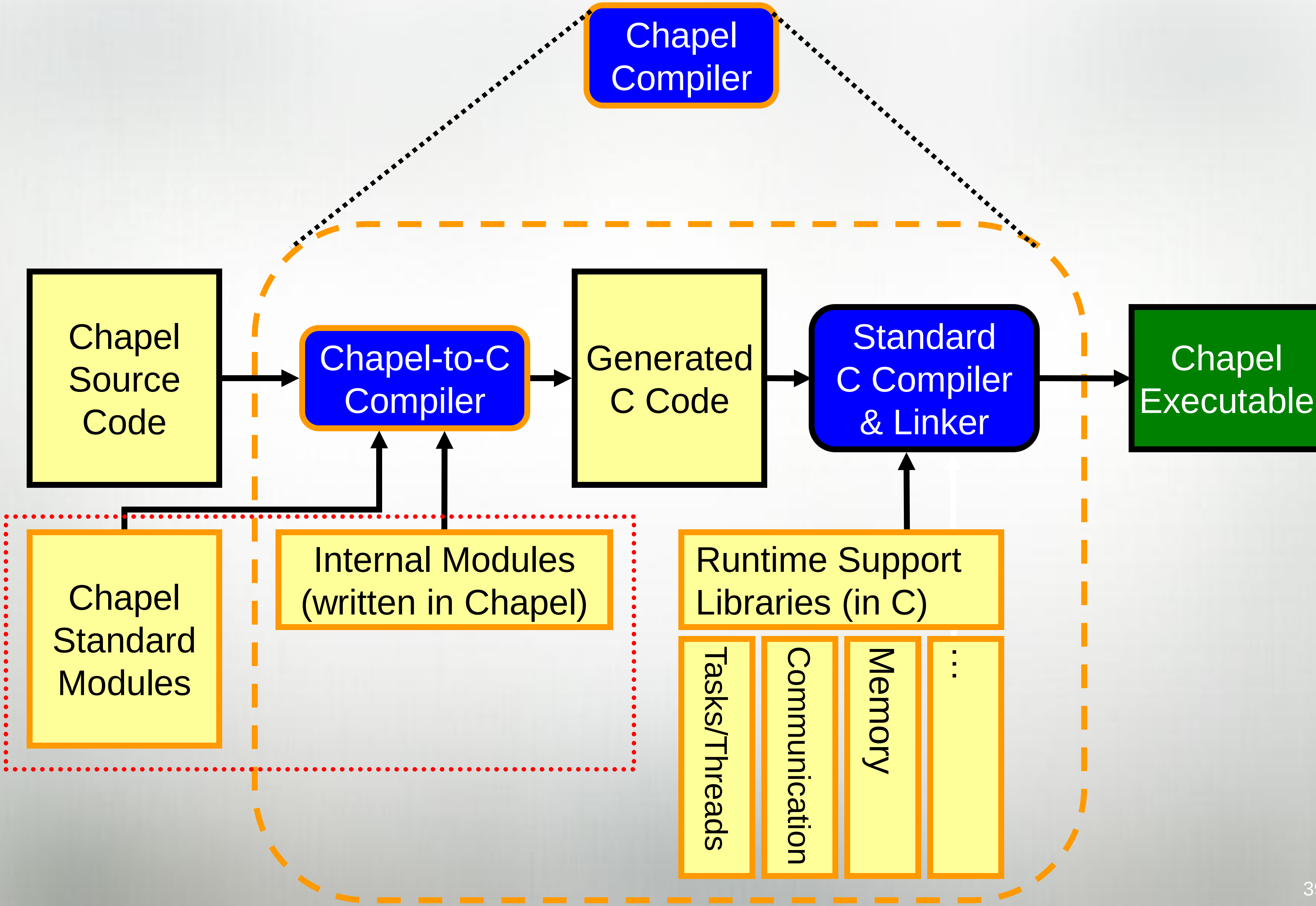
```
class LocalBlockArr {
  type eltType;            // generic field for array element type
  var data: […] eltType;   // a non-distributed array of values
}

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);
  }
}
```

Data Parallelism

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



Data Parallelism

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: Layouts and Distributions

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, …

Descriptors for Layouts

Domain Map

Represents: a domain map value

Generic w.r.t.: index type

State: domain map representation

Size: Θ(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: Θ(1) → Θ(*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: Θ(*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 Distributions

	Domain Map	Domain	Array
Global one instance per object (logically)	**Role:** Similar to layout's domain map descriptor	**Role:** Similar to layout's domain descriptor, but no Θ(#indices) storage **Size:** Θ(1) → Θ(#locales)	**Role:** Similar to layout's array descriptor, but data is moved to local descriptors **Size:** Θ(1)
Local one instance per locale per object (typically)	**Role:** Stores node-specific domain map parameters	**Role:** Stores node's subset of domain's index set **Size:** Θ(1) → Θ(*#indices / #locales*)	**Role:** Stores node's subset of array's elements **Size:** Θ(*#indices / #locales*)

Things I didn't Quite Get To

- 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

THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel-info@cray.com http://sourceforge.net/projects/chapel/

Multiresolution: Chapel's Domain Map Strategy

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
    - $\Rightarrow$ semantics as portable as the domain maps themselves

Q: How Can Chapel Succeed When HPF Failed?

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