Hewlett Packard
Enterprise

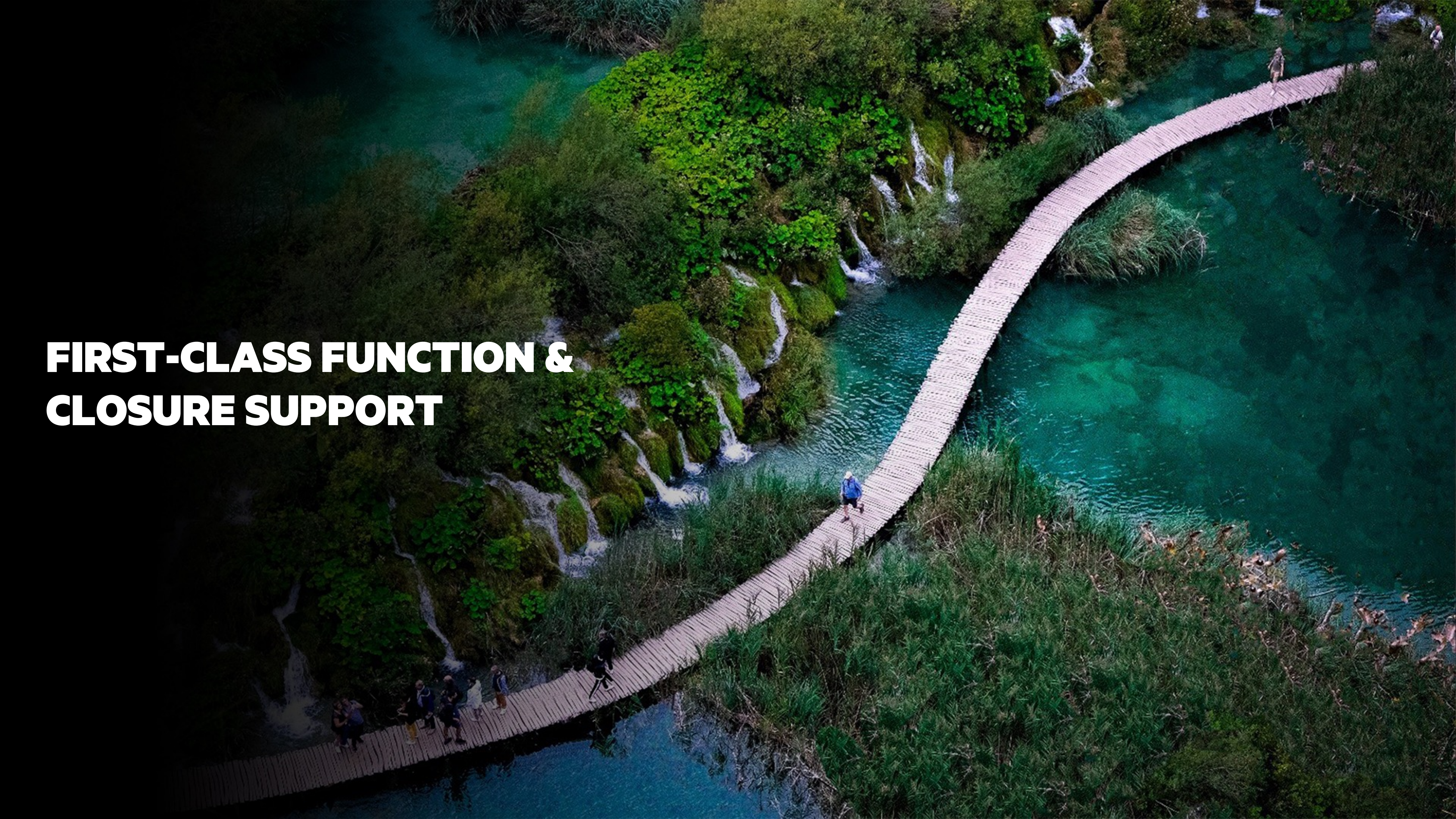
CHAPEL RELEASE NOTES, 1.25.1 / 1.26.0: ONGOING EFFORTS

Chapel Team

December 9, 2021 / March 31, 2022

OUTLINE

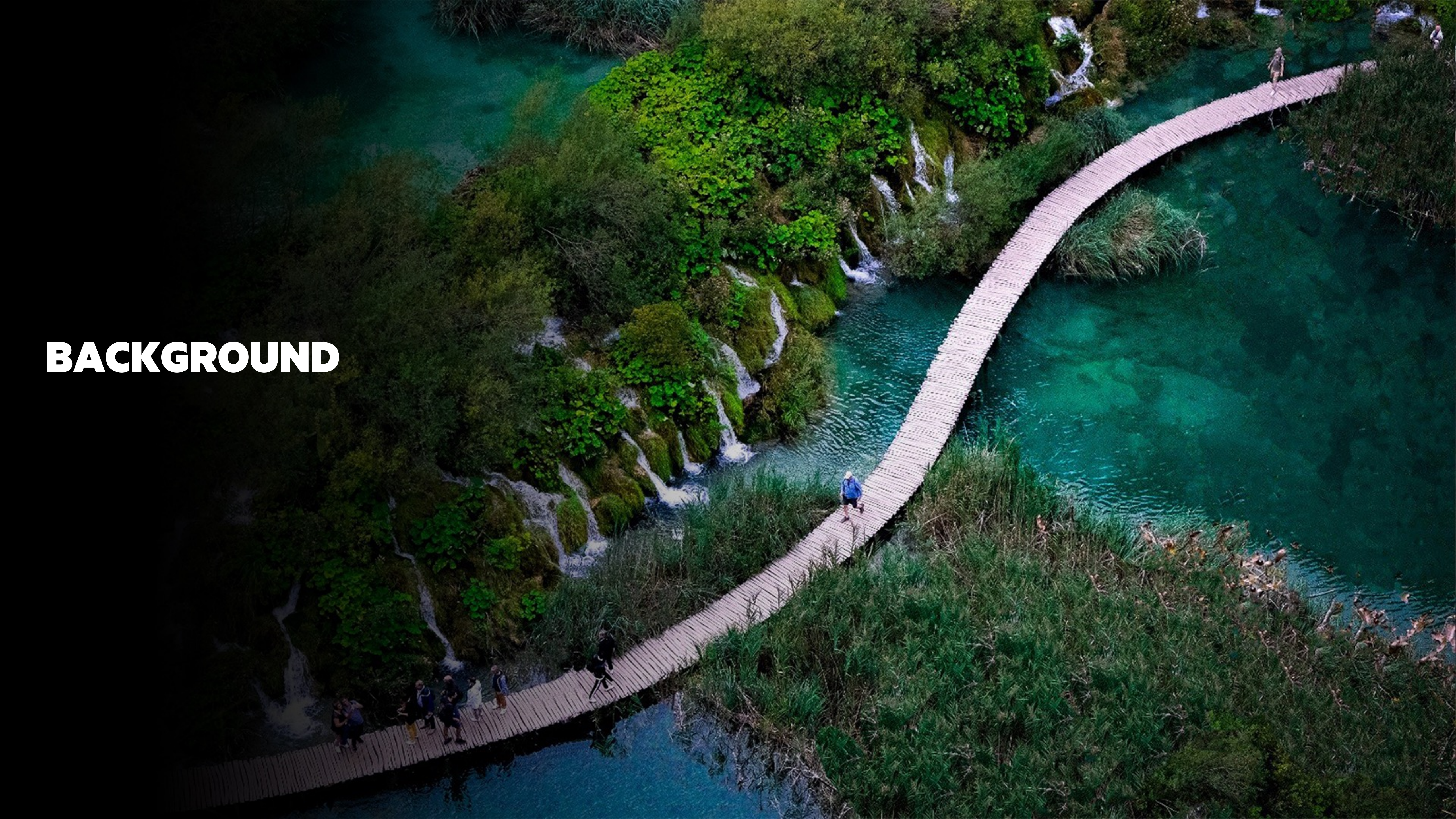
- First-Class Fns & Closures
- Serial/Par./Dist. Collections
- Special Method Naming
- GPU Support
- Compiler Rework



FIRST-CLASS FUNCTION & CLOSURE SUPPORT

FIRST-CLASS FUNCTION & CLOSURE SUPPORT

- Background
- Function Types & Values
- Closure Types & Values
- Summary



BACKGROUND

FIRST-CLASS FUNCTION & CLOSURE SUPPORT

Background: Informally Defining Terms

First-class-function (FCF): A function that can be passed around like other values

Closure: A first-class-function that refers to outer variable(s)

Outer variable: A local-scope variable defined in an enclosing scope

Capture: Storing a reference or copy of an outer variable to refer to within a closure

FIRST-CLASS FUNCTION & CLOSURE SUPPORT

Background: Motivation

- First-class-functions (FCFs) and closures have been identified as a desirable language feature
  - Can provide a useful means of abstraction and code reuse

- Consider the signature of the 'map.update()' method from the standard modules

```
// Apply 'updater' to the value associated with map key 'k'
proc ref map.update(const ref k: map.keyType, updater) { ... }
```

- Ideally, the 'updater' argument would accept a closure for simplicity and flexibility
  - Right now, Chapel only supports a limited form of FCF, which cannot represent a closure
  - Instead, for cases like this, users can pass a "function object" (a callable record/class) to emulate a closure
    - Requires the 'update()' method to leave the 'updater' formal generic

FIRST-CLASS FUNCTION & CLOSURE SUPPORT

Background: Motivation

Users can pass a "function object" (a callable record) to emulate a closure

- The boilerplate required often renders this undesirable

```
// Steps required to make use of 'map.update()' without closure support
proc example() {
  use Map;
  var m: map(int, int);
  m.put(0, 0);
  // Might have to define a new "function-object" for each call to 'update'!
  record myMapUpdater {
    var x = 0;
    proc this(const ref k, ref v) { v = x; }
  }
  var updater = new myMapUpdater(8);
  m.update(0, updater);
  writeln(m.get(0));   // Prints '8'
}
```

FIRST-CLASS FUNCTION & CLOSURE SUPPORT

Background: FCF Support Available Today

- Create a FCF from a named function

```
proc f() { writeln('hello'); }
var fcf = f;
```

- Create a FCF from an anonymous function with lambda syntax

```
var fcf = lambda() { writeln('hello'); };
```

- Create function types with the builtin 'func()' type function

```
type F = func(int, bool, real);    // a type representing a function that takes int & bool args, returning a real
```

- Nested functions can refer to outer variables

```
proc f() {
  var x = 0;
  proc g() { x += 1; }    // Here 'g' refers to 'x'
  g();
  writeln(x);             // Prints '1'
}
```

- FCFs that reference outer variables are not supported today
  - Crashes the compiler

FIRST-CLASS FUNCTION & CLOSURE SUPPORT

Background: Discussion Roadmap

The rest of the slides will propose adding the following features:

- Function types and values
- Closure types and values
  - And how they differ from function types
- Rules for converting functions to closures (and vice versa)
- ‘param’ functions and closures

- We will not discuss anonymous functions in these slides
  - Though we intend to support them

FUNCTION TYPES & VALUES

FUNCTION TYPES & VALUES

Problems with Existing Function Types

Today, function types are constructed via a built-in function

```
type T = func(int, bool, real);   // a type representing a function that takes int & bool args, returning a real
```

- Problems exist with the current strategy
  - The type constructor function cannot include argument intents or return intents
  - The call to 'func' is arguably harder to read
    - Return type not immediately apparent
    - Name 'func' not conventional

- Ideally there would be dedicated syntax to construct function types
  - Consider the syntax for defining a function:

```
proc foo(x: int, y: bool): real { ... }
```

FUNCTION TYPES & VALUES

Add Syntax to Construct Function Types

Proposal: Introduce syntax to construct function types

```
type T = proc(int, bool): real;
```

- Immediately easier to read and reason about
  - What argument types does the above function type take? What type does it return?
  - Syntax mirrors procedure definition:

```
proc foo(x: int, y: bool): real { ... }
```

- Argument and return intents are necessarily part of function types
  - Form part of the ABI for making function calls

```
type A = proc(ref int, int): int;
type B = proc(int, int): int;
assert(A == B);                     // False, different intents means different types!
```

FUNCTION TYPES & VALUES

What is Included in a Function Type?

Including argument and return intents in function types seems obvious

- What about formal argument names? Do we allow them to appear at all?

```
type T1 = proc(x: int): int;   // Are these two types equal?
type T2 = proc(int): int;
```

- A function marked 'inline'?

```
type T3 = inline proc(): void;   // Are these two types equal?
type T4 = proc(): void;
```

- What about…
  - A function marked with 'throws'?
  - A function with a 'where' clause?
  - Default argument values?

FUNCTION TYPES & VALUES

Components Included in a Function Type

- **Proposal**: Named arguments may be included, but they affect the type of the function

```
assert(proc(x: int): void == proc(int): void);    // False!
```

- **Proposal**: The 'inline' keyword is not part of a function's type
  - Calls to a FCF created from an inlined function are not guaranteed to be inlined

- **Proposal**: Default argument values are not part of function types

```
proc foo(in x: int=0): real;
writeln(foo.type);                  // Prints 'proc(in int): real'
```

- **Proposal**: The ‘throws’ keyword affects the type of the function

```
assert(proc(): void throws == proc(): void);    // False!
```

- **Proposal**: First-class-functions cannot have 'where' clauses

FUNCTION TYPES & VALUES

Type Inference and Function Overloading

What happens when a FCF's initialization expression is an overloaded function?

- Which overload of 'f' is selected when creating 'fcf'?

```
proc example() {
  proc f(x: int) {}
  proc f(x: real) {}
  var fcf = f;           // Which overload of 'f' is selected?
  fcf();
}
```

- **Proposal**: It is an ambiguity error unless an explicit type is specified

```
proc proposal() {
  var fcf: proc(x: int): void = f;    // OK, selects 'proc f(x: int)'
  fcf();
}
```

FUNCTION TYPES & VALUES

Outer Variables

Often the functions that users want to pass around do not refer to outer variables

- For example, such cases should be able to be passed to an external C function pointer argument

```
proc f(x: int): void { writeln(x); }

// This extern function takes a function type, equivalent to 'void (*fn)(int)'
extern proc c_call_func(fn: proc(int): void): void;

// Pass the Chapel function 'f' to C
c_call_func(f);
```

Proposal: Function values cannot refer to outer variables

- Simplifies their semantics and implementation
- Allows them to be used interchangeably with C function pointers

CLOSURE TYPES & VALUES

CLOSURE TYPES & VALUES

Refresher on Function Objects

Recall that users can currently pass a "function object" (a callable record) to emulate a closure

- The boilerplate required often renders this undesirable

```
// Steps required to make use of 'map.update()' without closure support
proc example() {
  use Map;
  var m: map(int, int);
  m.put(0, 0);
  // Might have to define a new "function-object" for each call to 'update'!
  record myMapUpdater {
    var x = 0;
    proc this(const ref k, ref v) { v = x; }
  }
  var updater = new myMapUpdater(8);
  m.update(0, updater);
  writeln(m.get(0));   // Prints '8'
}
```

CLOSURE TYPES & VALUES

Add Support for Closures

Proposal: Add closure types and values to empower use cases like 'map.update()'

- Pass a closure to 'update' instead of a "function object"

```
proc example() {
  var m: map(int, int);
  m.add(0, 0);
  var x = 8;

  // Here the nested function 'f' refers to the outer variable 'x'
  proc f(k: m.keyType, ref v: m.valType) { v = x; }
  m.update(0, f);          // Use 'f' to 'update' the value associated with key '0'
  writeln(m.getValue(0));  // Prints '8'
}
```

- Unlike function values, closure values can refer to outer variables
  - We must decide how outer variables are stored

CLOSURE TYPES & VALUES

Why Not a Single Type for Functions and Closures?

Both the compiler and user may need to know if a function is a closure

- If there is only one type, how can they tell?

```
proc example() {
  var x: int;
  proc f() {}
  proc g() { writeln(x); }
  var fcf1 = f;
  var fcf2 = g;
  assert(fcf1.type == fcf2.type);   // Are the types of 'f' and 'closure' the same?
}
```

- If the types of ‘fcf1’ and ‘fcf2’ are the same, then they should be useable in the same ways
  - But because ‘fcf2’ is a closure, it may have additional constraints for how it is used
    - E.g., we could return ‘fcf1’ trivially, but perhaps not ‘fcf2’

CLOSURE TYPES & VALUES

Separate Types for Functions and Closures

Proposal: Provide separate types for functions and closures

- Closure types can refer to outer variables, while function types cannot

```
proc example() {
  var x: int;
  proc f() {}
  proc g() { writeln(x); }
  var fcf1 = f;
  writeln(fcf1.type:string);   // Prints 'proc(): void'
  var fcf2 = g;
  writeln(fcf2.type:string);   // Prints 'closure(): void'
}
```

Proposal: Introduce a way to construct a closure type (perhaps new syntax)

- E.g., as used in the following examples…

```
type T = closure(): void;
```

CLOSURE TYPES & VALUES

Converting Function Types to Closure Types

It would be inconvenient if function values could not be passed to formals of type ‘closure’

- This would require duplication of code or leaving the formal generic

```
proc example() {
  proc takeAndCall(fn: closure(): void) { fn(); }
  var x: int;
  proc f() {}
  proc g() { writeln(x); }
  takeAndCall(g);   // OK
  takeAndCall(f);   // Error
}
```

- **Proposal**: Allow function types to implicitly convert to closure types

```
takeAndCall(f);   // OK, converts to closure type
```

CLOSURE TYPES & VALUES

Converting Closure Types to Function Types

Proposal: In certain cases, allow closure types to implicitly convert to function types

- Must know statically that the closure does not capture outer variables

```
proc example() {
  proc f() { writeln('Hello world!'); }

  proc takeAndCall(fn: proc(): void) { fn(); }

  const fcf1: closure(): void = f;
  takeAndCall(fcf1);   // OK, initializer is known to be 'f'
}
```

CLOSURE TYPES & VALUES

Calling Closures in Performance-Critical Code

In this example we consider a closure called in a loop. We might like the call ‘fn(i)’ to be inlined.

```
proc example() {
  var x: int;
  inline proc g(arg: int) { return arg + x; }

  proc callMe(fn: closure(int): int) {
    var sum = 0;
    for i in 1..n do sum += fn(i);
  }

  var fcf = g;   // Create a closure out of 'g'
  callMe(fcf);
}
```

CLOSURE TYPES & VALUES

Param Closures

Proposal: Introduce 'param' function and closure types to optimize calls at compile-time

- The value passed to ‘fn’ must be known at compile-time

```
proc example() {
  var x: int;
  inline proc g(arg: int) { return arg + x; }

  proc callMe(param fn: closure(int): int) {
    var sum = 0;
    for i in 1..n do sum += fn(i);   // Replace 'fn' with call to 'g' at compile-time
  }

  param fcf = g;   // Create a 'param' FCF out of 'g'
  callMe(fcf);
}
```

CLOSURE TYPES & VALUES

Returning Closures

When returning a closure, it is not obvious how to store outer variables

- E.g., store by value, or on heap with a reference count

```
proc example() {
  proc makeClosure() {
    var x: int;
    proc g() { writeln(x); } // How does 'x' get stored here?
    return g;
  }
  var fcf = makeClosure(); // What does this print?
  fcf();
}
```

- **Proposal**: Make returning closures an error for the foreseeable future
  - Support only closures used strictly within their lexical scope, for now

CLOSURE TYPES & VALUES

Assigning Closures

Closure assignment is potentially subject to the same issues as returning a closure

- Still uncertain of how outer variables should be stored

```
proc example() {
  var fcf: closure(): void = ...;   // Assume initialization
  proc makeAndAssignClosure() {
    var x: int;
    proc g() { writeln(x); }
    fcf = g;   // How does 'x' get stored here?
  }
  fcf();
}
```

- **Proposal**: Prevent such "up-scope" assignments for closures with lifetime checking
  - Modify the lifetime checker as needed to handle such cases

CLOSURE TYPES & VALUES

Always Capture Outer Variables by Reference?

Is there ever a time where outer variables should *not* be captured by reference?

- Does the following code print '1' or '0' for 'x'?

```
proc example() {
  var x: int;
  proc g() { x += 1; }
  var fcf = g;
  fcf();
  writeln(x);    // What is printed here?
}
```

Proposal: Closures will only capture by reference for the foreseeable future

- This seems to align with user intuition
- Could add syntax to explicitly capture if we are uncertain, e.g.,

```
var fcf = g with (in x);
```

SUMMARY

FIRST-CLASS FUNCTIONS & CLOSURE SUPPORT

Summary

We propose several directions for improving FCF support

- Add a function type and a separate closure type
  - Define rules for implicitly converting between functions and closures
  - Explore errors and edge-cases for the two types
  - Add 'param' functions and closures to enable inlining

- Closures always capture outer variables by reference
  - Prevent closures from being returned for now
  - Restrict closure assignment with lifetime checking

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

COLLECTIONS OUTLINE

- Chapel Today
- Survey of Other Languages
- Histogram Example
- Other Examples
- Side-by-side with Arrays
- Wrap-up

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Background

- Chapel has four standard modules that provide collection types: List, Set, Map, and Heap

- Goal: create a plan for supporting distributed versions of these collections
  - None of these modules have support for distributed implementations today
    - But it seems reasonable to add them in the future

  - Distributing these types across locales could impact their interfaces
    - For each type, we want the interface to be as similar as possible across the serial, parallel, and distributed variants
    - So, don't want to make interface decisions without understanding the impact
    - The 2.0 effort has motivated examining the current interfaces with these concerns

- Decided to look at:
  - what Chapel has today
  - what other languages do
  - examples of potential use cases, especially for distributed versions

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Guiding Questions

- Should we have a single type per collection or distinct serial/parallel/distributed variants of the types?

- How much should the interface differ for serial, parallel, and distributed versions?

- How similar should our approach be to that of our arrays?

- What should a user be able to do with a parallel or distributed collection?
- What shouldn't a user be able to do?

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Current Proposal

- Should we have a single type per collection or distinct serial/parallel/distributed variants of the types?
  - *Based on precedent from other languages, lean towards separate implementation*

- How much should the interface differ for serial, parallel, and distributed versions?
  - *For users' benefit, should be as similar as possible*

- How similar should our approach be to that of our arrays?
  - *For users' and implementers' benefit, should be similar*
    - *e.g., for distributed lists, it makes sense to use distribution patterns like 'BlockCyclic' and 'Cyclic'*
      - *'Block' may make less sense, given the dynamically changing sizes of lists*

- What should a user be able to do with a parallel or distributed collection?
- What shouldn't a user be able to do?
  - *Proposal details are later in these slides*

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Tradeoffs between single vs. multiple types

type declarations:

- single type would require different arguments to specify the desired flavor (e.g., ‘list(parSafe=true)’)
- multiple types would require changes to the type name itself (and possibly additional arguments as well)

uses of variable:

- hopefully, interfaces are similar enough to not necessitate changes when becoming more parallel, distributed
  - parallel implementations should be superset of serial, and distributed should be subset of parallel

generic programming:

- want some way to write a routine that can take any flavor of a type, whether through ‘list(?)’, an interface, or ???

default task intents:

- serial versions should be ‘const ref’ by default, which seems appropriate to avoid races
- ‘ref’ intent seems more appropriate for parallel/distributed versions

implementation architecture:

- using single type could entangle implementation details within the code
- multiple types could lead to repetition of common code or the need for shared code across flavors

CHAPEL TODAY

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

What Chapel Has Today

- All four collections have a 'parSafe' field and a hidden lock field
  - Only collections created with 'parSafe==true' are suitable for concurrent access
  - When 'parSafe==true', all operations (except most of the iterators) acquire the hidden lock
- Also have a ‘ConcurrentMap’ package developed by Garvit Dewan and Louis Jenkins
  - There's a distributed version, too, but it hasn't been contributed to the project
- There are also ‘DistributedBag’ and ‘DistributedDeque’ implementations in the package directory

SURVEY OF OTHER LANGUAGES

OTHER LANGUAGES’ COLLECTIONS

C++

- Collections provide assurance of avoiding data races in certain situations but not others
  - E.g., parallel additions to a map are fine but parallel adjustments to the same index are not

- Boost library provides separate Distributed Property Map
  - No distributed set, list, or heap
  - Also provides distributed "adjacency list" and distributed queue

- Berkeley Container Library (BCL) provides a distributed hash map
- Can also get "distributed" maps, sets and lists via Hazelcast library
  - "Distributed" for Hazelcast is more along the lines of cloud computing, sharding, consistency models, etc.
    - This is different than BCL's and Chapel's definition of "distributed"

OTHER LANGUAGES' COLLECTIONS

Java

- Set and List have a parallel stream that can be obtained to perform computations
  - These computations are not expected to modify the collection itself, but don't prevent it
- The Map implementations don't support parallel streams
  - But might be able to use on result of 'keySet', 'values' and 'entrySet' methods for limited operations
- Couldn't find a Heap type implementation
- Additionally, there are separate types to support parallelism in the java.util.concurrent package
  - E.g., there is a ConcurrentMap with what appears to be all the methods of a serial map
- Can get "distributed" maps, sets, and lists via Hazelcast library
  - See note about Hazelcast from C++ slide

OTHER LANGUAGES' COLLECTIONS

Swift

- Supports Set, List, and Dictionary (map equivalent?) and a BinaryHeap type
  - No mention of parallelism or concurrency in their documentation
    - Set and Dictionary distinguish between operations on mutable versus immutable versions, but that's it

- Community-contributed Swift Concurrent Collections package is relatively new

- Language's concurrency docs only mention control from user's side

- User forum suggests steering away from thread-safe dictionaries as a concept:
  - "It adds unavoidable overhead to every single access, and doesn't solve the higher-level synchronization problems of the code that's using the dictionary"

OTHER LANGUAGES' COLLECTIONS

Go

- No built-in set or list type that we could find
- Maps are a default type
  - Lots of packages for separate concurrent map implementations
- Heaps available as a package
  - No concurrent or parallel heaps
- Can get "distributed" maps, sets and lists via Hazelcast library
  - See note about Hazelcast from C++ slide

OTHER LANGUAGES' COLLECTIONS

Rust

- Provides all four collection types, some with multiple implementations (e.g., HashSet and BTreeSet)
- Has crate Rayon for data parallelism
  - Provides parallel iterators for each standard collection
- There's an alternate concurrent hash map implementation on its own
- Has Github crate Amadeus for distributed iterators
  - Though it's a little hard to see what exactly is supported there

OTHER LANGUAGES' COLLECTIONS

Summary of Supported Types

		C++	Java	Swift	Go	Rust
Map	parallel	ish	limited	no	(many)	iterators (Rayon)
	distributed	Boost, Hazelcast, BCL	Hazelcast	no	Hazelcast	iterators (Amadeus)
Set	parallel	ish	iterators	no	no set type	iterators (Rayon)
	distributed	Hazelcast	Hazelcast	no	Hazelcast	iterators? (Amadeus)
List	parallel	ish	iterators	no	no	iterators (Rayon)
	distributed	Hazelcast	Hazelcast	no	Hazelcast	iterators (Amadeus)
Heap	parallel	ish	no heap type	no	no	iterators (Rayon)
	distributed	no	no heap type	no	no	iterators? (Amadeus)

OTHER LANGUAGES' COLLECTIONS

Summary of Approaches

	C++	Java	Swift	Go	Rust
Separate type for serial/parallel/distributed	🟨 Separate type for distributed	🟪		🟪	🟪
Same type for serial/parallel/distributed		🟨 Parallel streams			🟨 parallel and distributed iterators
No support	🟨 Not much parallelism		🟪		

Key

- 🟨 Language falls under this category, but with caveats
- 🟪 Language falls under this category

OTHER LANGUAGES’ COLLECTIONS

Guiding Questions, revisited

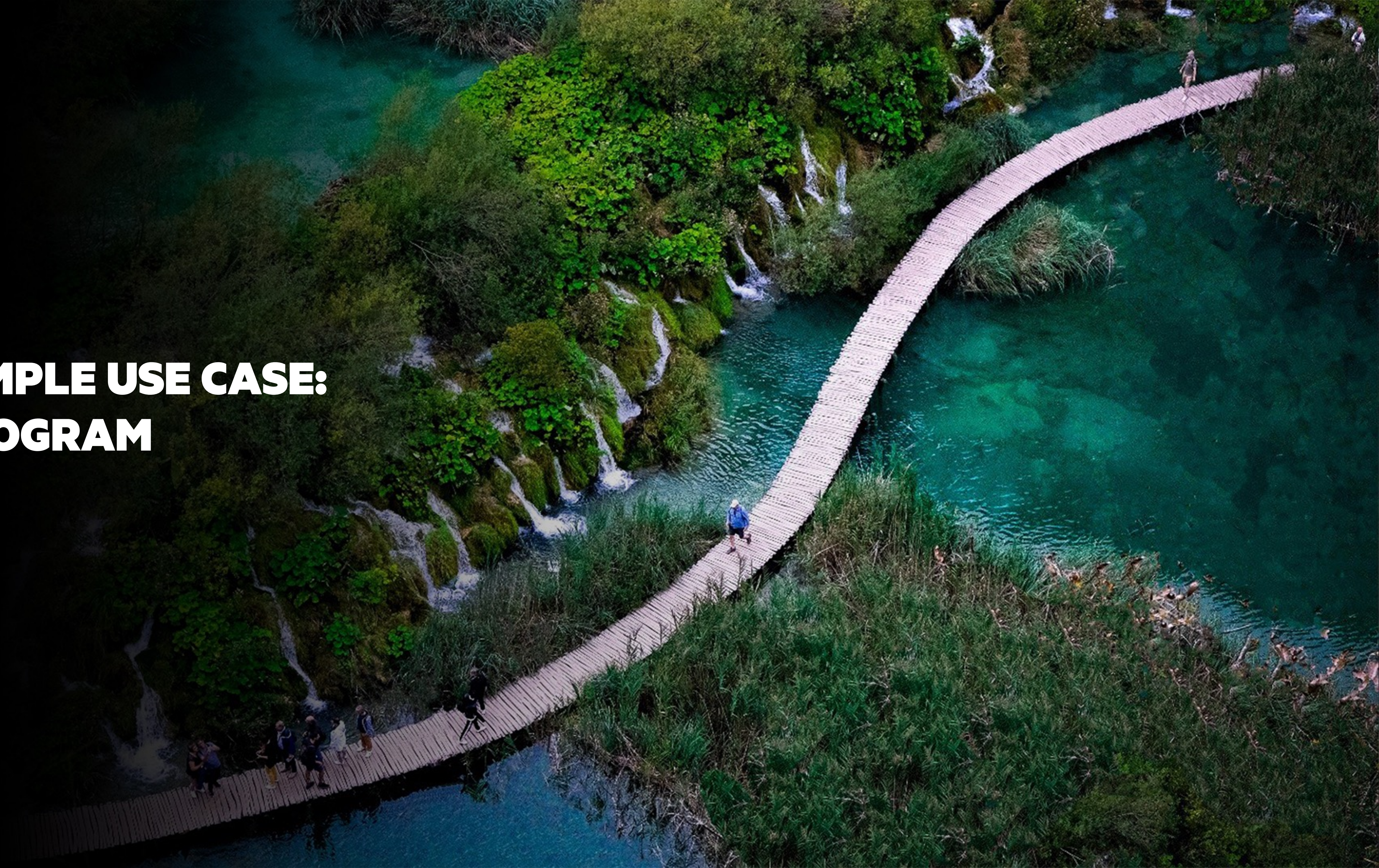
- Should we have a single type per collection or distinct serial/parallel/distributed variants of the types?
  - *Other languages are handling this using a separate type or more limited fashion*
  - *There are many caveats with these implementations, but it is still valuable to know about them*

- How much should the interface differ for serial, parallel and distributed versions?
  - *Seems like the interface is mostly similar*

- How similar should our approach be to that of our arrays?

- What should a user be able to do with a parallel or distributed collection?
- What shouldn't a user be able to do?
  - *We'll talk more about these when looking through the examples*

EXAMPLE USE CASE: HISTOGRAM



HISTOGRAM EXAMPLE

Introduction

- Consider a distributed histogramming program:

```
// suppose 'hist' is a distributed map, 'input' is a distributed array
forall key in input {
  hist[key] += 1;
}
```

- This could cause a race
  - 'hist[key]' and '+=' are two separate operations
    - Without care, some action could come between them
  - A 'key' added by another task could cause a resize, making the reference to 'hist[key]' invalid
    - Even if adding keys doesn't impact already existing keys, a parallel thread could still remove the one being referenced
    - This is known as *reference invalidation*
  - see issue #19102

- How can we know when it is valid to aggregate updates?

HISTOGRAM EXAMPLE

Reference Invalidation Approaches

- The next set of slides will discuss possible solutions to the reference invalidation issue, using
  - First-Class Functions
  - Context Managers
  - Critical Sections
  - Compound Operators

- The impact of all four will be compared

HISTOGRAM EXAMPLE

First-Class Functions (FCFs)

- Could pass in a FCF object to an updater method, which will maintain safety of the element
  - Code snippet from PR #19554 (which mocks up a Distributed Map) and current Map implementation

```
proc DistMap.update(k: keyType, updater) throws {
  const (locIdx, mapIdx) = keyToLocaleAndMapIdx(numLocales, numLocalMaps, k);
  on targetLocales[locIdx] {
    ref map = localMaps[mapIdx];
    map._enter(); defer map._leave();    // 'map' will have 'parSafe==true'
    ref val = map[k];                    // simplification
    updater(k, val);
  }
}
forall key in input {
  hist.update(key, lambda(key, ref element) { element += 1; });
}
```

- Can insert aggregation handling into update method without changing user interface

HISTOGRAM EXAMPLE

Context Managers

- Could use a context manager to control updates

```
forall key in input {
  manage hist.element(key) as elt {
    elt += 1;
  }
}
```

```
proc eltWrap.enterThis() ref eltType {
  impl.lock();
  return this.elt;
}

proc eltWrap.leaveThis(in error: owned
                              Error?) throws {
  impl.unlock();
  if error then throw error;
}
```

- Issue: how to handle the case when the element does not already exist in the map?
- Naïve approach: create, then lock on every element
- Better approach: aggregate updates and apply them on each locale in bulk
  - Compiler would add aggregation implicitly when the ‘forall’ expression supports a certain API to hook onto

HISTOGRAM EXAMPLE

Critical Sections

- Add new syntax (e.g. 'sync'?) that prevents races on some aspect of data structure, executing on locale
  - Could be identical to context manager
    - Relies on compiler hook methods to use with a particular type
    - Explicit different syntax would make intended behavior clearer
  - Additionally, compiler will recognize presence within 'forall'
    - Can implicitly insert aggregation
    - When last statement in forall body, can enable forall-unordered optimization

```
forall key in input {
  sync hist[key] {
    hist[key] += 1;
  }
}
```

- Issues: new syntax, relies heavily on compiler support

HISTOGRAM EXAMPLE

Compound Operators

- Support definition of compound operator for both access and update

```
// Exact syntax TBD
proc DistMap.this(idx: int).+=(rhs: eltType) { … }
…
forall key in input {
  hist[key] += 1;    // Compiler recognizes combination of 'this' and '+=' as matching the compound operator definition
}
```

- Solves reference invalidation correctness problem by avoiding returning a 'ref'
- Issue: doesn't help with performance from remote calls
  - Could insert aggregation into compound operator definition explicitly

HISTOGRAM EXAMPLE

How To Avoid Reference Invalidation

Strategy	Advantages	Disadvantages
FCFs	- Collection controls access to element, enabling aggregation w/o user-facing changes - Enables applying whole operation to element at once w/o user locking element	- User must learn about FCFs - Might be tricky to make method general enough - FCFs are not yet mature in Chapel - User must adjust code to use
Context Manager	- Simple API for collections to implement - Natural use of language feature - Can aggregate updates as part of enterThis/leaveThis interface with later flush	- User must adjust code to use - Relies heavily on compiler support
Critical Sections / 'sync'	- Hook methods should be applicable to other collections - Provides aggregation and access to optimizations - Syntax makes user intention obvious	- New keyword/syntax - Relies heavily on compiler support
Compound Operators	- No change to user code (if correct compound combination defined)	- Doesn't solve performance problem - Might result in increased code duplication with individual portions of compound operators - No difference in safe and unsafe user code, relies on definition of individual compound operator

HISTOGRAM EXAMPLE

Aggregation Approaches

- The next set of slides will discuss how to determine if aggregation is possible
- It will cover:
  - Data Structure Memory Consistency Model
  - Forall-unordered Optimization
  - Explicit Aggregator Objects

- The impact will be compared

HISTOGRAM EXAMPLE

Data Structure Memory Consistency Model

- Instead of immediately logging changes, collection itself will batch its updates to other locales
  - And flush changes regularly, including for each memory fence operation
  - Like cache-remote optimization, and intended to hook into Memory Consistency Model explicitly

- User interface unchanged
  - Though may want to call flush explicitly

```
// User code same, but behavior of '[]' access will be different
forall key in input {
  hist[key] += 1;
}
```

- Allows optimizations and tuning to the specific collection and implementation

- Risk: implementation might be directly tied to collection type
  - Less potential for code re-use across different collections than an outside aggregator would provide, for instance

HISTOGRAM EXAMPLE

Forall-unordered Optimization

- Compiler can analyze how ‘forall’ iterations update a data structure
  - Checks if sole modification to data structure is the last statement in the body

- User interface unchanged

```
// User code same, order unimportant and can be rearranged for performance
forall key in input {
  hist[key] += 1;
}
```

- Strategies for handling reference invalidation could thwart current compiler analysis
  - We can teach compiler to recognize them

HISTOGRAM EXAMPLE

Explicit Aggregator Objects

- Aggregator specifies information needed to batch changes
  - Waiting to communicate changes across locales minimizes overhead of remote calls
    - Instead of waiting for 10 remote calls, make 1 remote call with 10 updates in it, for example

- Can store aggregator in collection, calling it as part of implementing updates
  - Example with FCFs

```
// Aggregator logs update, or idx and how to make update, then performs them at later time
proc DistMap.update(k: keyType, updater) throws {
  agg.add(k, updater);
}
// User code sends updater to apply to key when ready to perform updates
forall key in input {
  hist.update(key, updater=lambda(key, ref element) { element += 1; });
}
```

- Or can call aggregator explicitly in user code

HISTOGRAM EXAMPLE

Summary

Strategy	Advantages	Disadvantages
Data Consistency Model	- No change to user code (except occasional flush) - Enables read-ahead and prefetch in helping updates - Could enable specialized optimizations for collection	- Solution may be too tied to individual data structure
Forall-unordered Optimization	- No change to user code - Less work when developing collections	- Relies heavily on compiler support, can be thwarted - Might need to provide hints to compiler in data type
Explicit Aggregator	- Enables code reuse across collections - Impact on user code depends on strategy chosen - Aggregation is explicitly defined and visible in code	- Explicit control is a maintenance burden - May require explicit tuning - Applicability to multiple collections avoids collection-specific optimizations

OTHER EXAMPLES

OTHER EXAMPLES WE'VE BEEN LOOKING AT

Patterns:

- Creating a distributed map from a distributed set
- Merging two distributed maps
- Label propagation pattern from CHIUW 2016

Design questions (and our current thinking)**:**

- Should equality checks between sets, maps, lists take implementation / locality into account? (no)
- How should 'list' users specify where elements are stored? (using block-cyclic-style arguments at creation time)
- How should 'set'/'map' users control locality? (by specifying an optional hash from keys to locales)
- Should creating a distributed map from a distributed set preserve locality? (optionally, yes)
- If two maps are merged, is locality preserved? (when their inter-locale mappings are identical, yes)
- Are distributed heaps important to have in the 2.0 timeframe? (don't currently think they're on the critical path)

SIDE-BY-SIDE WITH ARRAYS

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Side-by-side

- This next section will demonstrate how code is expected to evolve from serial to distributed
  - It uses ‘map’ as an example, though ‘set’ or ‘list’ could be reasonably substituted
  - Each slide will show a similar adjustment to a program using arrays
    - This should demonstrate that adjustments are at least somewhat in keeping with how arrays are defined today

  - Note: collections must worry about parallel safety more than arrays
    - Collections are more likely to be resized, moving their elements in memory

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Side-by-side

- Serial code

```
var dom = {0..<100};
var arr: [dom] int;

…

for idx in dom {
  arr[idx] += 1;
}
```

```
var m = new SerialMap(keyType=int, valType=int);

…

for key in m.keys() {
  m[key] += 1;
}
```

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Side-by-side

- Same code, but adjusted to run in parallel

```
var dom = {0..<100};
var arr: [dom] int;

…

forall idx in dom {
  arr[idx] += 1;
}
```

```
var m = new ParallelMap(keyType=int, valType=int);

…

forall key in m.keys() {
  elt += 1;
}
```

- Could consider keeping parallel and serial handling in the same type and distributed map in a separate type
  - Map creation will need an adjustment either way:

```
var m = new map(keyType=int, valType=int);
var m = new map(keyType=int, valType=int, parSafe=true);
```

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Side-by-side

- Same code, but adjusted to be distributed

```
var dom = {0..<100}
  dmapped Cyclic(startIdx=0);
var arr: [dom] int;
…

forall idx in dom {
  arr[idx] += 1;
}
```

```
var m = new DistributedMap(keyType=int, valType=int,
                           dist=myLocaleHasher);
…

forall key in m.keys() {
  elt += 1;
}
```

- Exact interface for how to distribute a map is a design choice
  - It would almost certainly need to be specified at creation time

WRAP-UP

SERIAL/PARALLEL/DISTRIBUTED COLLECTIONS

Wrap-up

- The precedent for distributed data structures comes with a lot of caveats
  - Most support tends to be either independent types with the same interface or more limited iterators
  - These caveats don't mean we shouldn't do it necessarily, just that it may be charting new territory
    - As a parallel language by design, we can play a leadership role here
    - We lean toward not providing a distributed Heap, though, at least initially

- There are some collections that may benefit from explicit distribution patterns like we have for arrays
  - E.g., BlockCyclic, or Cyclic for the List type
  - And others that may benefit from a version with user-controlled distribution strategies, for greater flexibility
    - Though we may not provide one immediately

- Finding the balance between providing an optimized interface and a familiar one may take some care
  - Where possible, our goal should be to unify as much as possible

SPECIAL METHOD NAMING

SPECIAL METHOD NAMING
Background

- Chapel currently has a small, but nontrivial, number of special methods
  - some cases are supported on [many | most | all] types
  - some cases are intended to be invoked by the [compiler | user | both]
  - some cases are defined by [Chapel | the user | Chapel when the user does not]

- Current special methods:
  - **intrinsic Chapel queries:** '.type', '.locale'
  - **initialization/deinitialization:** '.init', '.postinit', '.init=', '.deinit', '.complete'
  - **direct access / iteration:** '.this', '.these'
  - **I/O:** '.readThis', '.writeThis'
  - **context management:** '.enterThis', '.leaveThis'
  - **hashing:** '.hash'

- The possibility of new special methods being added later raises concerns for backward-compatibility
  - if a user is already using a method with that name and signature, it's likely to cause breaking changes

SPECIAL METHOD NAMING

This Effort

- Studied current special methods to identify categories and patterns
  - found a surprising lack of symmetry
- Discussed paths forward to avoid breaking changes
  - reserving more keywords
  - reserving a specific naming/formatting convention
  - introducing a new identifier convention for such cases

SPECIAL METHOD NAMING

Definitions and Uses

- Who implements these methods?
  - **Chapel implements, user cannot:** '.type', '.locale', '.complete'
  - **User can implement, otherwise Chapel will/may:** '.init', '.init=', '.deinit', '.read/.writeThis', '.hash'
  - **User can implement, but need not:** '.postinit', '.this', '.these', '.enter/.leaveThis'
- Who calls these methods?
  - **Designed for user to call:** '.type', '.locale', '.complete'
  - **Only Chapel can call:** '.init=', '.deinit'
  - **Intended for either Chapel or users to call:** '.init', '.hash'
  - **User may call, but typically (?) wouldn't:** '.postinit', '.this', '.these', '.read/.writeThis', '.enter/.leaveThis'
- How strict are the signatures?
  - **completely or partially constrained:** '.init=', '.deinit', '.postinit'
  - **unconstrained, where new overloads enable new capabilities:** '.init', '.this'
  - **constrained, but users can create/call other versions:** '.hash', '.these', '.read/.writeThis', '.enter/.leaveThis'

SPECIAL METHOD NAMING

Naming Schemes

- Current naming schemes:
  - **Reserved word:** ‘.type’, ‘.locale’, ‘.init=’, ‘.this’
  - **direct/straightforward:** ‘.init’, ‘.postinit’, ‘.deinit’, ‘.hash’
  - **“doThis”:** ‘.readThis’, ‘.writeThis’, ‘.enterThis’, ‘.leaveThis’
  - **(too?) cute/clever:** ‘.this’, ‘.these’

- Alternate naming schemes:
  - **reserve everything?** big hammer; some names don’t feel keyword-y; doesn’t help with future-proofing
  - **reserve more?** ‘.init’, ‘.deinit’, ‘.postinit’ (?) seem like potential candidates
  - **use “doThis” more?** ‘.hashThis’, ‘accessThis’, ‘.iterateThis’
  - **use an alternative to “doThis”?**
    - “Alice in Wonderland”: ‘.writeMe’, ‘.accessMe’, …
    - formatting conventions: ‘.__hash__’, ‘._hash’, ‘.hash_’, … (note that, unlike Python, some cases are intended for users to call)
    - non-identifier formatting: ‘.|hash|’, ‘.<hash>’, ‘.:hash:’, ‘.(hash)’, ‘.*hash*’, ‘.-hash-’, ‘.@hash’, ‘.$hash’, ‘.hash~’, ‘.hash$’, …
    - see issues #19038 and #19050 for further discussion

SPECIAL METHOD NAMING

One Potential Approach

- Preserve (non-cute) cases that rely on keywords:
  - .type, .locale, init=

- Reserve additional keywords and restrict user definitions of—and calls to—them:
  - init, deinit, postinit

- Unify on ‘.doThis’ for all other cases

```
– old:                               – new:
  .hash()                              .hashThis()
  .this()       /  .these()            .accessThis() / .iterateThis()   // or ‘.traverseThis()’?
  .complete()                          .completeThis()                  // (?)
```

SPECIAL METHOD NAMING

Another Potential Approach

- Preserve (non-cute) cases that rely on keywords:
  - .type, .locale, init=

- Reserve additional keywords and restrict user definitions of—and calls to—them:
  - init, deinit, postinit

- Select a non-identifier formatting convention for other cases and use simplified names
  - e.g., if we were to use ‘-name-’ as the convention:

–old: – new:

```
.hash()                          .-hash-()
.this()      / .these()          .-access-()   / .-iterate-()
.readThis()  / .writeThis()      .-read-()     / .-write-()
.enterThis() / .leaveThis()      .-enter-()    / .-leave-()     // or -exit-()?
.complete()                      .-complete-()                  // (?)
```

SPECIAL METHOD NAMING

Status and Next Steps

Status:

- we’re beginning to have some acceptable paths forward

Next Steps:

- complete consensus-building
  - get input from users
- identify special formatting for non-reserved names
- implement new names and deprecate old ones

GPU SUPPORT

GPU SUPPORT

Background

- We have been working on adding native GPU support to Chapel
  - One of the most sought after features among users
  - Earlier collaborations with academia and industry influenced the design

Releases

- **1.23:** Design effort and discussions started
- **1.24:** Can use non-user-facing features to generate GPU binaries for Chapel functions and launch them
- **1.25:** Can natively generate Chapel functions from order-independent loops and launch them on GPUs
- ***1.26:*** *Much bigger portion of the base language can now execute on the GPU, significant design progress*

GPU SUPPORT OUTLINE

- GPUs in 1.26: Overview
- New Capabilities
- Locale Model Design
- Ongoing Work
- Summary & Priorities

GPU SUPPORT IN 1.26: AN OVERVIEW WITH IDIOMS

GPU SUPPORT

Where We Were in Chapel 1.25

```
on here.getChild(1) {
  var A, B, C: [1..n] real;
  const alpha = 2.0;

  forall b in B do b = 1.0;
  forall c in C do c = 2.0;

  forall (a, b, c) in zip(A, B, C) do
    a = b + alpha * c;
  // or
  forall i in A.domain do
    A[i] = B[i] + alpha * C[i];

}
```

- Arrays are allocated in unified memory
- Scalars are allocated on the function stack
  - So, they are on host memory

Promotion (e.g., ‘B = 1.0’) still executed on host, so used explicit loops instead

All these foralls executed on GPU

GPU SUPPORT

Some of the New Abilities in 1.26

```
coforall subloc in 1..<here.getChildCount() do on here.getChild(subloc) {

  var A, B, C: [1..n] real;
  const alpha = 2.0;


  B = 1.0;
  C = 2.0;

  A = B + alpha * C;

}
```

More than one GPU per locale can be used

Promoted expressions are launched as kernels

GPU SUPPORT

Idioms in 1.26 – Copy data to GPU, launch a kernel, copy result back

Note: There may be better idioms for expressing the same operation in the future

```
var A: [0..<n] int;
```

```
on here.getChild(1) {
  var AonGPU = A;
```

- ‘AonGPU’ will be allocated on the Unified Memory
- The host array ‘A’ will be copied in bulk

```
  AonGPU += 1;
```

Promotion will execute on GPU

```
  A = AonGPU;
}
```

Result is copied back to the host

NEW CAPABILITIES

NEW CAPABILITIES
Overview

Features & Implementation

- Most functions can be called from GPU kernels
- Promoted expressions run on GPUs
- More than one GPU can be used within the node
- Kernels can have nested blocks (if, for, etc.)

- Compiler primitives for
  - block synchronization
    - i.e., '__syncthreads()' in CUDA
  - shared memory allocation
    - i.e., '__shared__' variables in CUDA

- GPU binary is now embedded in the executable
- Better checking for environment variables

Design & Explorations

- New locale model design for GPUs
  - More intuitive interface
- Started investigating libomptarget
  - Portable alternative to current CUDA-based runtime
- Design discussions on user-facing forall configs
  - A way to query task IDs, set block size etc.

Outreach

- Talk at SIAM PP22 in minisyposium
  - *Code Generation and Transformation in HPC on Heterogeneous Platforms*
- Talk proposal submission at CHIUW 2022

NEW CAPABILITIES

Multiple GPUs in a Locale

- In 1.25.x, the runtime was hard-coded to use the first GPU only
  - Even if there were multiple GPUs in the node, you could only use one

```
on here.getChild(2) { ... }    // would behave the same as here.getChild(1) – the first GPU would be used
```

- In 1.26, the runtime can handle multiple devices within the node
  - CUDA contexts are created for each device at startup, and used based on the sublocale ID
  - A multi-device version of simplified stream where all devices do HPCC Stream individually:

```
coforall sublocID in 1..here.getChildCount() do on here.getChild(sublocID) {
  var A, B, C: [1..n] int;

  B = 1; C = 2;
  A = B + alpha * C;
}
```

NEW CAPABILITIES

Idioms in 1.26 – Static work-sharing between CPU and the GPU(s) in the same node

Note: There may be better idioms for expressing the same operation in the future

```
var A: [0..<n] int;

// assign half the work to CPU, the rest to GPUs. Assume divisibility
const numGPUs = here.getChildCount()-1;
const cpuSize = n/2;
const gpuSize = (n/2)/numGPUs;

cobegin {
  A[0..<cpuSize] += 1;

  coforall subloc in 1..numGPUs do on here.getChild(subloc) {
    const myShare = cpuSize+gpuSize*(subloc-1)..#gpuSize;

    var AonThisGPU = A[myShare];
    AonThisGPU += 1;
    A[myShare] = AonThisGPU;
  }
}
```

Compute 'gpuSize' and 'cpuSize' based on the decomposition

CPU works on its part

Two concurrent tasks

GPUs work on their part and copy the result back

NEW CAPABILITIES

Promoted Expressions in Kernels

- In 1.25.x, promoted expressions would run on the host even if they were in a GPU block

```
on here.getChild(1) {
  var A, B: [1..n] int;
  A = 1;       // you'd need an explicit foreach/forall to run as a kernel
  B = 2 * A;  // same here
}
```

Recall that in the current GPU locale model, child 0 is the CPU, >0 are GPU(s). This interface will improve.

- In 1.26, the promoted expressions will be run on the device as kernels
  - Allows HPCC Stream to be expressed in the most succinct form and be run on the device

```
on here.getChild(1) {
  var A, B, C: [1..n] int;

  B = 1; C = 2;           // both will be run as kernels
  A = B + alpha * C;    // as with other Chapel promotions, this will be a zero-copy
                        // operation that is run on the device
}
```

NEW CAPABILITIES

Nested Blocks in Kernels

- In 1.25.x, the loop body must have been a basic block
  - Constructs like ‘if’ and ‘for’ inside kernels were not tested and not working

- In 1.26, such blocks can appear inside the loop body that is turned into a kernel

```
on here.getChild(1) {
  foreach i in A.domain {    // will successfully be launched as a kernel in 1.26
    if i%2 == 0 then         // whereas, this ‘if’ would cause compiler crashes in 1.25
      A[i] = -i;
    else
      A[i] = i;
  }
}
```

NEW CAPABILITIES

Calling Other Functions from Kernels

- In 1.25.x, a loop was "GPU eligible" if it was
  - Order-independent (e.g., 'forall' or 'foreach')
  - In a user-defined module
  - Free of any function calls that are not inlined
  - Only using primitives that are "fast" and "local"
    - "fast" means "safe to run in an active message handler"
    - "local" means "doesn't cause any network communication"
- In 1.26, mostly removed the "free of any function calls" and "user-defined module" restrictions
  - Function calls are allowed if the function meets the above criteria and is non-recursive
  - Functions that are potentially called from GPU kernels are generated twice
    - One copy for the GPU and one for the CPU
  - Calling functions that are not "GPU eligible" results in the loop executing on CPU

- Calling halting functions from kernels is still future work

NEW CAPABILITIES

Calling Other Functions from Kernels

```
config param n = 100;

on here.getChild(1) {
  var A: [0..<n] real = 1.0,
      B: [0..<n] real = 2.0;
      C: [0..<n] real;

  // now executes on the GPU
  forall i in 0..<n {
    C[i] = add(A[i], B[i]);
  }

  // recursive function call disqualifies GPU execution
  forall i in 0..<n {
    C[i] = recursive(A[i], B[i]);
  }
}
```

```
proc add(a: real, b: real) {
  return a + b;
}

proc recursive(a: real, b: real) {
  if a < 10 then
    return a + b;
  else
    return recursive(a/2, b*2);
}
```

NEW CAPABILITIES

Embedded GPU binary

Background:

- The Chapel compiler produces a ‘.fatbin’ file that packages PTX code for GPU kernels
  - This file was stored under the ‘--savec’ path or under ‘*execName*_gpuFiles’ if no path was specified
- Previously, this file was read at runtime whenever you launched at kernel
- This meant if the ‘.fatbin’ file was moved or removed there would be a runtime error

This effort: Embed fatbin data inside the executable

- The compiler first produces the ‘.fatbin’ to a temporary directory but then:
- A later pass reads the contents of this file and stores it into a global variable
  - This file is removed at the end of compilation
- The runtime reads this variable and loads the code onto the GPU before launching a kernel

Impact:

- The executable is now portable and can me moved without depending on an external file

NEW CAPABILITIES

‘alloc shared’ and ‘sync threads’ primitives

Background:

- GPU kernels launch with a fixed number of threads partitioned into blocks
- Shared memory is shared among threads within the same block
- Sometimes it is desirable to synchronize threads within a block
- One use pattern: populate shared memory, synchronize threads, do computation

This Effort:

- We introduce two new primitives that deal with shared memory and synchronizing
- *These are meant to be used internally by the compiler, not by the user*

```
__primitive("gpu allocShared", size);  // allocate into shared memory
__primitive("gpu syncThreads");        // synchronize block threads
```

- Creating language features that expose these primitives to the user is future work

NEW CAPABILITIES

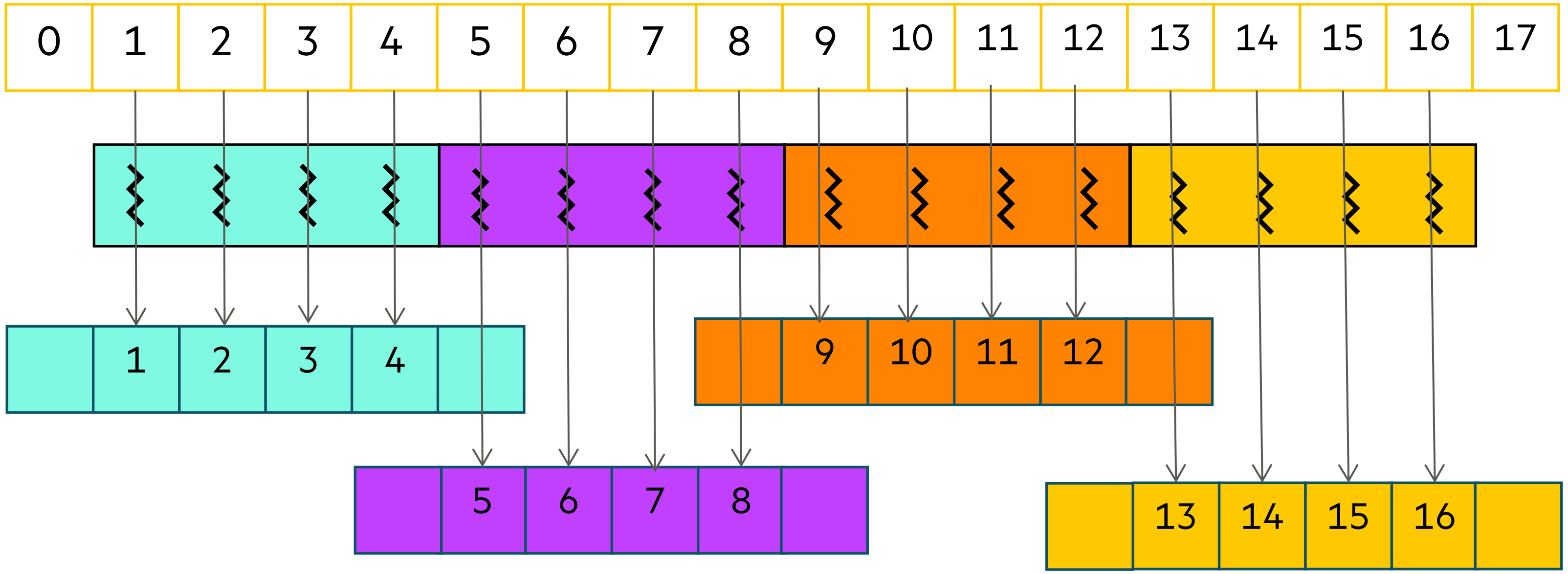
‘alloc shared’ and ‘sync threads’ primitives

- Update each element so:

```
A[i] =
   f(A[i-1], A[i],
     A[i+1]);
```

- Launch a kernel:
  - 16 threads blocks of 4
- Allocate a shared memory buffer
  - 6 elements to each block
- Copy element into its buffers

```
var A : [0..17] int;
```



NEW CAPABILITIES

‘alloc shared’ and ‘sync threads’ primitives

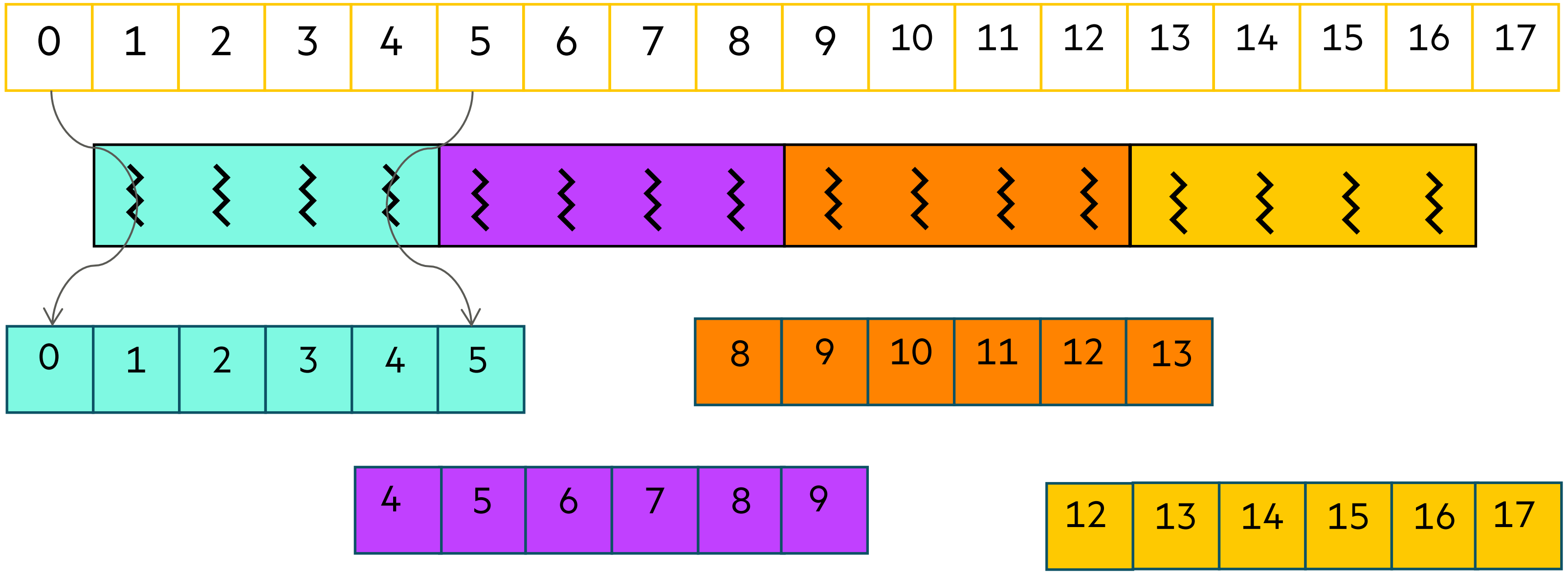
- Update each element so:

```
A[i] =
   f(A[i-1], A[i],
      A[i+1]);
```

- Launch a kernel:
  - 16 threads blocks of 4
- Allocate a shared memory buffer
  - 6 elements to each block
- Copy element into its buffers

- Copy first and last element
- Synchronize threads in a block

```
var A : [0..17] int;
```



NEW CAPABILITIES

‘alloc shared’ and ‘sync threads’ primitives

- Update each element so:

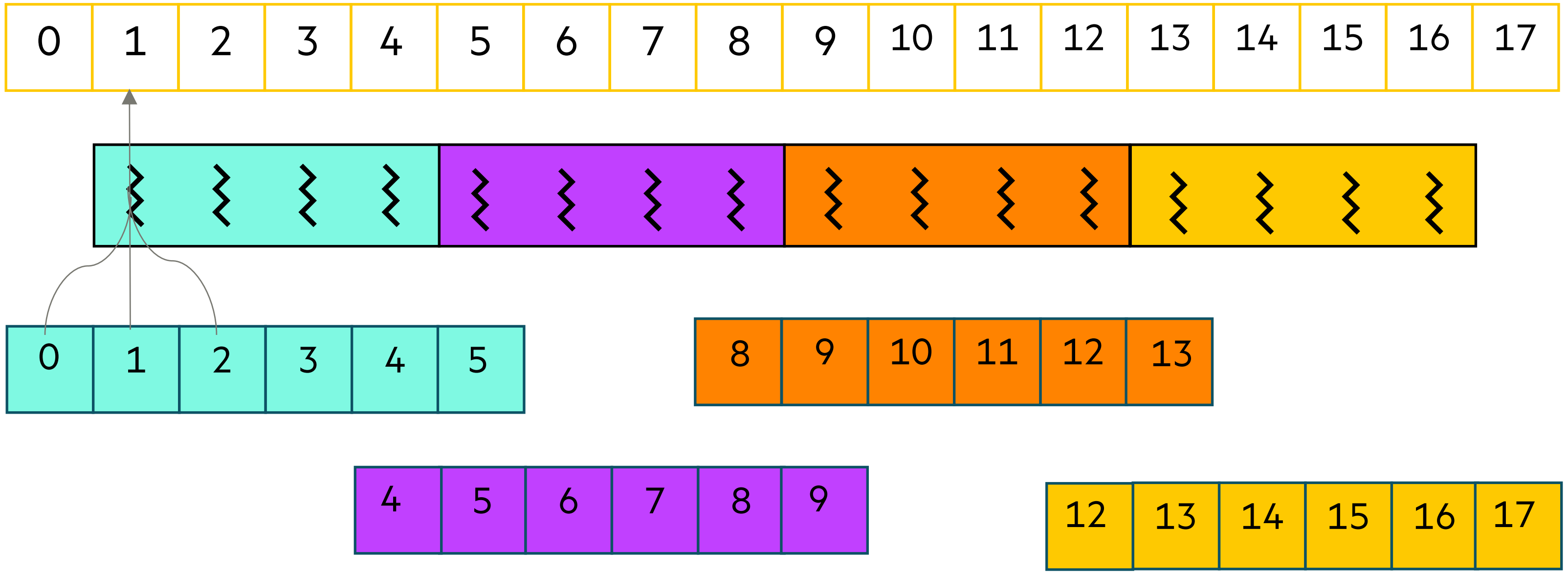
```
A[i] =
   f(A[i-1], A[i],
      A[i+1]);
```

- Launch a kernel:
  - 16 threads blocks of 4
- Allocate a shared memory buffer
  - 6 elements to each block
- Copy element into its buffers

- Copy first and last element
- Synchronize threads in a block

- Update each element A[i]

```
var A : [0..17] int;
```



NEW CAPABILITIES

‘alloc shared’ and ‘sync threads’ primitives

Here’s what it looks like in Chapel **please note:** this is not what we intend users to write; in the future you shouldn’t have to cast pointers or use primitives to get threadIdx; and sm should be representable with any domain

- Update each element so:

  ```
  A[i] =
    f(A[i-1], A[i],
      A[i+1]);
  ```

- Launch a kernel:
  - 16 threads blocks of 4
- Allocate shared memory buffer
  - 6 elements to each block
- Copy element into its buffers
- Copy first and last element
- Synchronize threads in a block
- Update each element A[i]

```
param N = 16, BLK_SIZE = 4;
on here.getChild(1) {
  var A : [0..N+1] = 0..N+1;
  foreach i in 1..N {
    var smVoidPtr = __primitive("gpu allocShared", BLK_SIZE+2)
    var sm = smVoidPtr : c_ptr(uint);
    var tid = __primitive("gpu threadIdx x") + 1;
    sm[tid] = A[i]
    if tid == 0 then sm[0] = A[i-1];
    if tid == BLK_SIZE then sm[BLK_SIZE+1] = A[i+1];

    __primitive("gpu syncThreads");

    A[i] = f(sm[tid-1], sm[tid], sm[tid+1]);
  }
}
```

NEW CAPABILITIES

‘alloc shared’ and ‘sync threads’ primitives

`__primitive("gpu allocShared", size)`

- returns a C ‘void*’ to a buffer of memory
- size parameter must be determined at compile time (like a param)
- in CUDA:

```
    __shared__ char s[size];
```

`__primitive("gpu syncThreads")`

- synchronizes threads within a block
- in CUDA:

```
    __syncthreads();
```

LOCALE MODEL DESIGN

LOCALE MODEL DESIGN

Background

- We define a locale as:

  “a Chapel abstraction for a piece of a target architecture that has processing and storage capabilities”
- We illustrate three locale models below
  - each model contains locales that may have sublocales underneath them
- Use ‘getChild()’ method to navigate from locale to sublocales

FLAT:

- Node 0
- Node 1

NUMA:

- Node 0
  - NUMA0
  - NUMA1
- Node 1
  - NUMA0
  - NUMA1

GPU:

- Node 0
  - CPU
  - GPU0
  - GPU1
- Node 1
  - CPU
  - GPU0
  - GPU1

LOCALE MODEL DESIGN

This Effort

- We've had discussions about:
  - how to adapt locale models to handle different (current and future) hardware
  - what changes we should make to the GPU locale model today

- Out of these discussions we've gotten:
  - **concrete short-term steps** to work on next: get rid of 'getChild()' method, get rid of CPU sublocale
  - a **conceptual shift**: that doing something 'on' a sublocale…
    - is enacting a policy for execution and memory allocation behavior
    - may result in different execution behavior for SIMT computation
  - a bunch of **open questions**

- We don't need a perfect design or to have all open questions settled to start making progress today

LOCALE MODEL DESIGN

Planned Changes for the GPU Locale Model

- Get rid of CPU sublocale in GPU locale model
  - because it’s no different than just using the Node locale

- Remove ‘getChild’ method
  - sublocale IDs should be opaque to the user
  - we would like the kind of sublocale (GPU, NUMA, etc.) to be indicated explicitly

- Have ‘getGpu’ and ‘numGpus’ methods instead
  - exact names to be decided

GPU:

- Node 0
  - ~~CPU~~
  - GPU0
  - GPU1
- Node 1
  - ~~CPU~~
  - GPU0
  - GPU1

LOCALE MODEL DESIGN

A Conceptual Shift

- Recall our definition for locale:
  "a Chapel abstraction for a piece of a target architecture that has processing and storage capabilities"
- How does this work with this example?

```
// Execution starts on the locale for node 0
var A: [1..N] int;          // Allocate in RAM
on here.getGpu(0) {         // On first GPU
  var B: [1..N] int;        // Allocate in unified memory
  doSomething();            // Perform on CPU (not in foreach loop)
  foreach i in 1..N {       // Perform on GPU
    B[i] = B[i] + 1;
  }
}
foreach i in 1..N {         // Perform computation on CPU
  A[i] = A[i] + 1;
}
```

The conceptual shift:
Think of the 'on' statement as imposing a policy, in this case where to perform SIMT computation and where to allocate arrays

This computation will not be done on the GPU even though we pass a GPU sublocale to the 'on' statement above

LOCALE MODEL DESIGN

Open Questions (1/2)

- What should happen when when the user attempts to access a GPU on a system that doesn't have one?

- There are two possibilities, we'd like the user to choose which one they want:
  - produce an error, or
  - fall back to using CPU

- Some options:
  - have an argument to specify behavior:
    - `on here.getGpu(0, errorIfUnavailable=true) { … }`
  - have two methods (one that has fallback behavior and another that errors):
    - `on here.getGpuIfAvailable(0) { … }`
    - `on here.getGpu(0) { … }`

LOCALE MODEL DESIGN

Open Questions (2/2)

- We want to support various policies:
  - compute on the CPU, allocate data into RAM
  - do SIMT-style computation on the GPU, do other computation on the CPU, allocate data into unified memory
  - in the future we may need/want more complicated policies
    - e.g., compute on a CPU within a specific NUMA domain and do SIMT computation on GPU and store data in FAM

- This raises some questions:
  - should there always be a one-to-one correspondence between sublocales and policies?
  - should we add syntax so ‘on’ statements explicitly show what policy they enact?

LOCALE MODEL DESIGN

Possible Directions

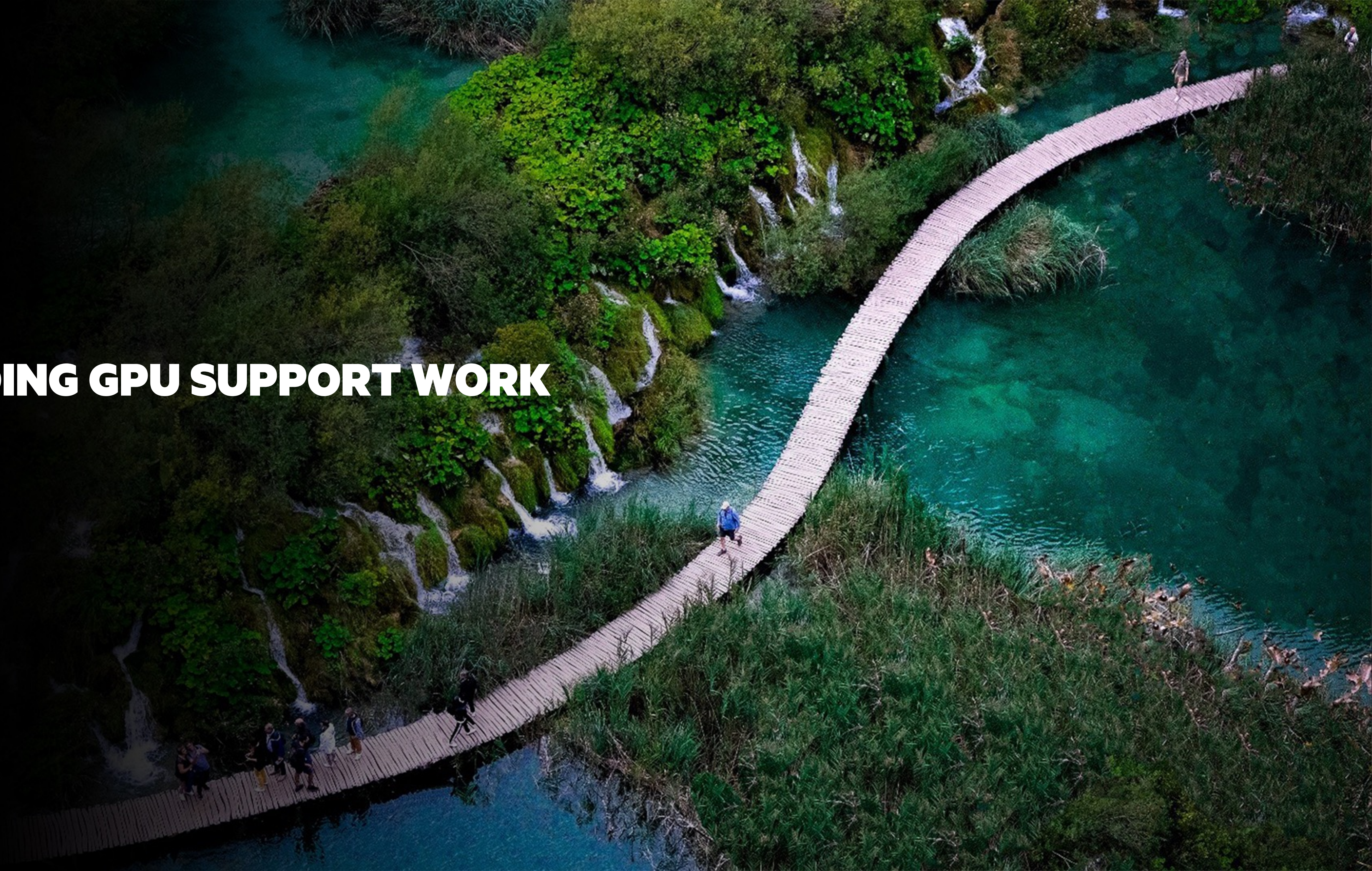
- One approach: keep one-to-one correspondence of sublocales-to-policies
  - but rename methods that return sublocales to make it more explicit what policy that sublocale implies:
    - instead of: `on here.getGpu(0) { … }`
    - do: `on here.withSimtComputationOnGpu(0) { … }`

- Another approach: introduce new syntax to ‘on’ that makes it explicit what policy is being enacted:
  - `on here withSimtComputationOn here.getGpu(0) { … }`

- Another approach: Create a new abstraction representing a policy that can be passed to ‘on’ statements:
  -
    ```
    on newPolicy(withNonSimtComputationOn = here.getNumaDomain(1),
                 withSimtComputationOn = here.getGpu(0),
                 withMemoryAllocationOn = Locales.gpuRamUnifiedMemory()) { … }
    ```

- One concern: if we introduce any new syntax/abstractions would they break forward compatibility?

ONGOING GPU SUPPORT WORK



ONGOING GPU SUPPORT WORK

Capabilities

Vendor portability

- In 1.26, only NVidia devices are supported
- We are investigating libomptarget as a portable runtime layer to support other GPU vendors

GPU-driven communication

- As a PGAS language, Chapel should support communication initiated from the device
- The runtime can use Unified Virtual Addressing (UVA) for allocations to be used with the communication layer
  - e.g., GASNet EX's memory kinds support enables registering UVA-based segments for communication

Using distributed arrays on GPUs

- Currently, 'targetLocales' can be used to map distributed domains onto GPUs
- However, lack of privatization onto device memory prevents using distributed arrays in kernels
- We are investigating whether we can replace privatization with remote value forwarding

ONGOING GPU SUPPORT WORK

Applications and Benchmarks

- CHAMPS
  - Existing effort on CHAMPS was based on interop and showed significant performance improvement
  - Achieving similar improvements using native GPU capabilities is one of our next application targets
- ChOp
  - **Ch**apel-based **Op**timization library that supports multiple GPUs on multiple nodes
  - Uses C interoperability to launch CUDA kernels
  - Initial experiments porting ChOp GPU kernels to Chapel look encouraging
- Existing Chapel benchmarks
  - LULESH, LCALS, PRK, …
- CORAL-2 (https://asc.llnl.gov/coral-2-benchmarks)
  - LAMMPS kernels, Pennant, Kripke
- SHOC Benchmarks (https://github.com/vetter/shoc)
  - Scalable HeterOgeneous Computing benchmark suite
- RAJAPerf (https://github.com/LLNL/RAJAPerf)

GPU SUPPORT

Longer-term Goals

- Multi-dimensional loops/kernels
- Task-/thread-private variables on GPU
  - Reductions
- Error handling
- Performance tuning

STATUS SUMMARY & PROPOSED PRIORITIES

GPU SUPPORT

A Rough Visualization of Progress

Maturity

Feature/ example

Sorted into three categories

	Design, investigate, prototype	In Chapel with language features	Works across nodes	Work across architectures	Performs well
Examples/benchmarks					
GPUAddNum, Stream					
Other benchmarks?					
Chapelize typical GPU programming model features					
Launching kernels w/ config.					
Query thread ID					
Block shared memory					
Multiple GPUs in a node					
GPU-ify existing Chapel features					
Locales / locale model					
Calling functions					
PGAS style puts/gets					
Distributions					
Promoted operators					
Reductions					

GPU SUPPORT

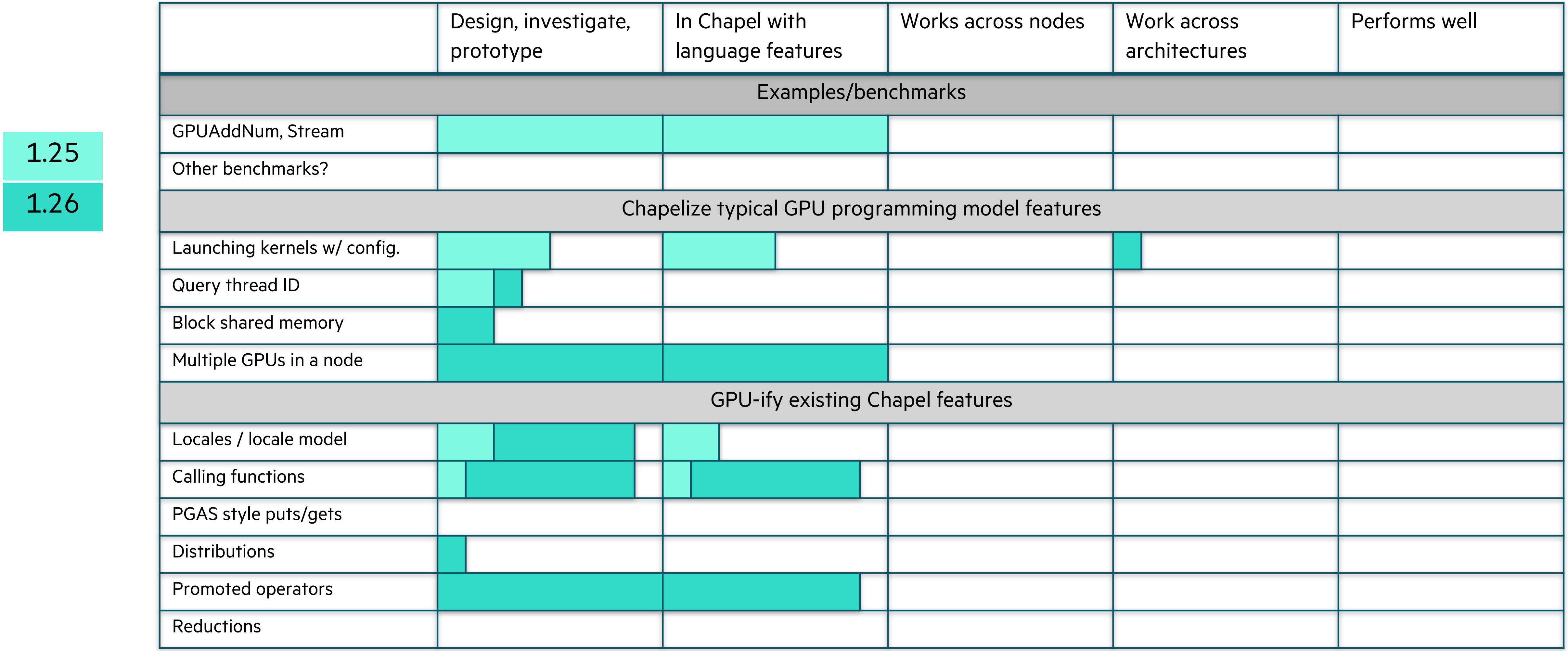
A Rough Visualization of Progress

1.25

	Design, investigate, prototype	In Chapel with language features	Works across nodes	Work across architectures	Performs well
Examples/benchmarks					
GPUAddNum, Stream	🟩	🟩			
Other benchmarks?					
Chapelize typical GPU programming model features					
Launching kernels w/ config.	🟩	🟩			
Query thread ID	🟩				
Block shared memory					
Multiple GPUs in a node					
GPU-ify existing Chapel features					
Locales / locale model	🟩	🟩			
Calling functions	🟩	🟩			
PGAS style puts/gets					
Distributions					
Promoted operators					
Reductions					

GPU SUPPORT

A Rough Visualization of Progress



🟩 1.25

🟦 1.26

	Design, investigate, prototype	In Chapel with language features	Works across nodes	Work across architectures	Performs well
Examples/benchmarks					
GPUAddNum, Stream	🟩🟩🟩🟩	🟩🟩🟩🟩			
Other benchmarks?					
Chapelize typical GPU programming model features					
Launching kernels w/ config.	🟩🟩	🟩🟩		🟦	
Query thread ID	🟩🟦				
Block shared memory	🟦				
Multiple GPUs in a node	🟦🟦🟦🟦	🟦🟦🟦🟦			
GPU-ify existing Chapel features					
Locales / locale model	🟩🟦🟦	🟩			
Calling functions	🟩🟦🟦🟦	🟩🟦🟦🟦			
PGAS style puts/gets					
Distributions	🟦				
Promoted operators	🟦🟦🟦🟦	🟦🟦🟦			
Reductions					

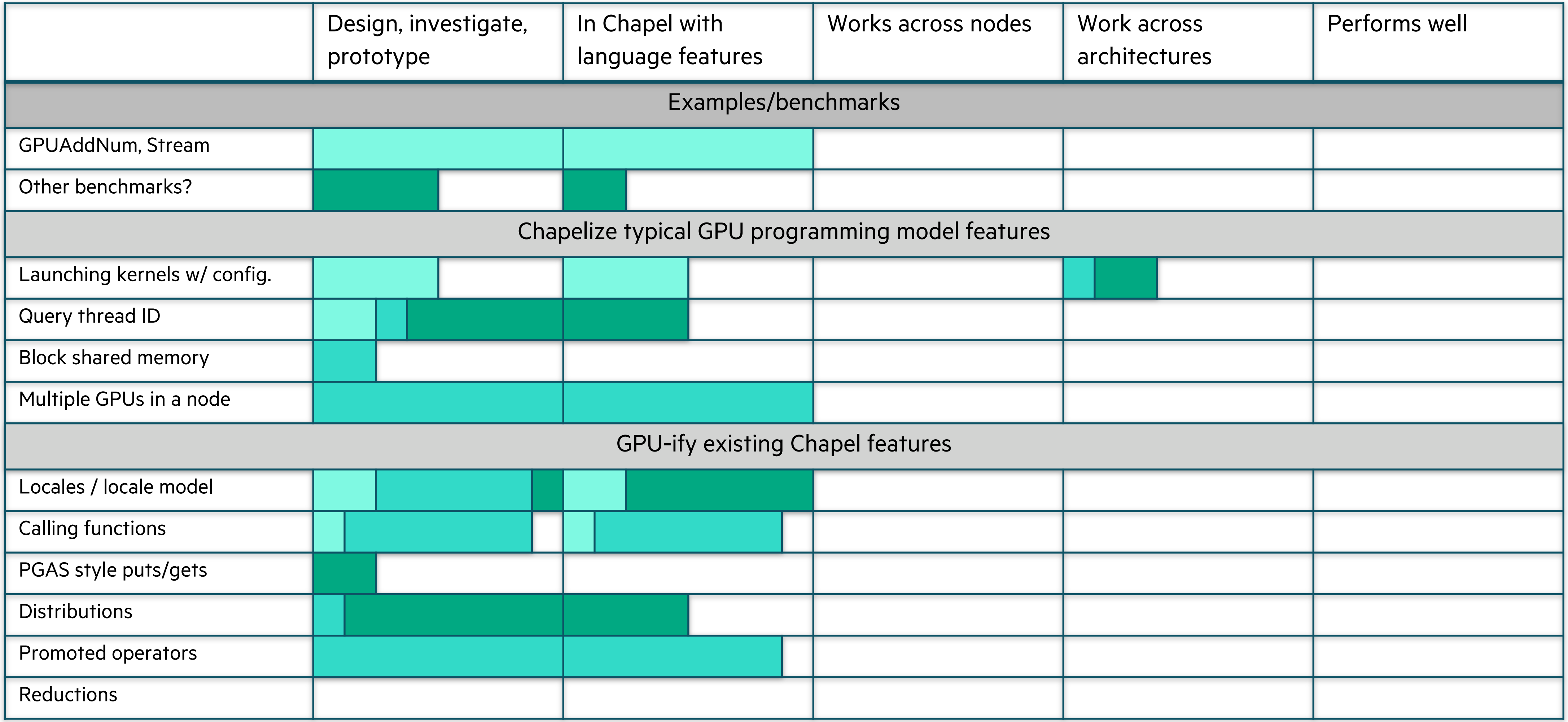
GPU SUPPORT

A Rough Visualization of Progress - Where We Want to Be Next

1.25

1.26

1.27



	Design, investigate, prototype	In Chapel with language features	Works across nodes	Work across architectures	Performs well
Examples/benchmarks					
GPUAddNum, Stream	1.25	1.25			
Other benchmarks?	1.27	1.27			
Chapelize typical GPU programming model features					
Launching kernels w/ config.	1.25	1.25		1.26 1.27	
Query thread ID	1.25 1.26 1.27	1.27			
Block shared memory	1.26				
Multiple GPUs in a node	1.26	1.26			
GPU-ify existing Chapel features					
Locales / locale model	1.25 1.26 1.27	1.25 1.27			
Calling functions	1.25 1.26	1.25 1.26			
PGAS style puts/gets	1.27				
Distributions	1.26 1.27	1.27			
Promoted operators	1.26	1.26			
Reductions					

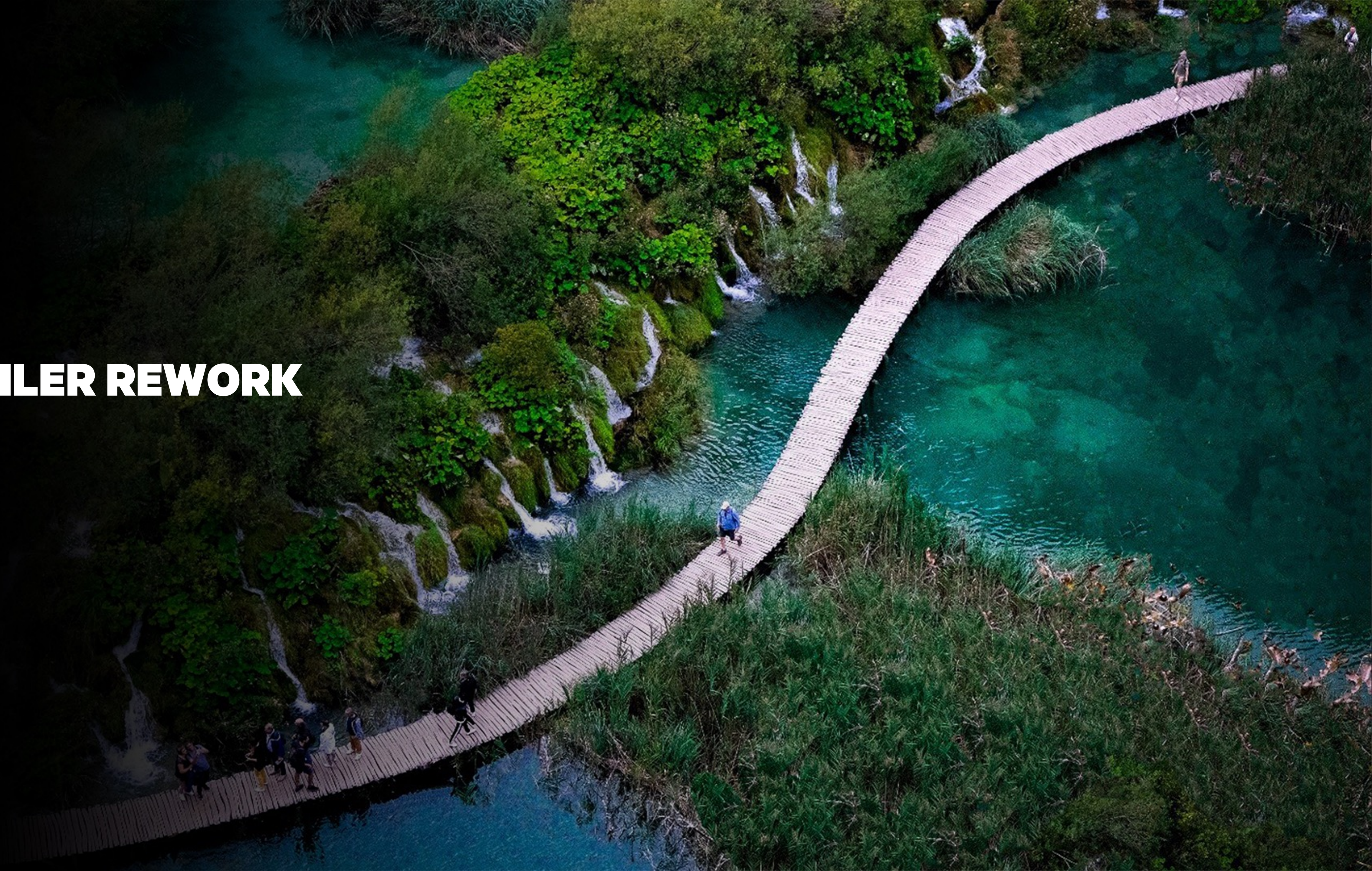
GPU SUPPORT
Summary

- 1.26 comes with many more features available to use with GPUs
  - Function calls
  - Nested blocks
  - Promoted expressions
  - Multiple GPUs in one locale

- We also have a plan for a new GPU locale model

- 1.27 priorities include
  - Vendor portability
  - Distributed memory capabilities
  - Application and benchmark studies

COMPILER REWORK



COMPILER REWORK OUTLINE

- Introduction and Motivation
- Separate Compilation
- Incremental Early Passes
  - Parsing to uAST
  - Resolving
- Later Passes
  - PassManager
  - Adjusting Passes
- Dyno-chpldoc
- Language Design Impact
- Summary

INTRODUCTION AND MOTIVATION

PROBLEMS WITH THE CURRENT CHAPEL COMPILER

Speed

- The current compiler is generally slow, and extremely so for large programs (~7s to 15 minutes)
- Large programs require complete recompilation whenever a change is made

Errors

- For incorrect programs, the compiler frequently displays only some of the errors at a time
- Compilation errors can be hard for users to understand and resolve

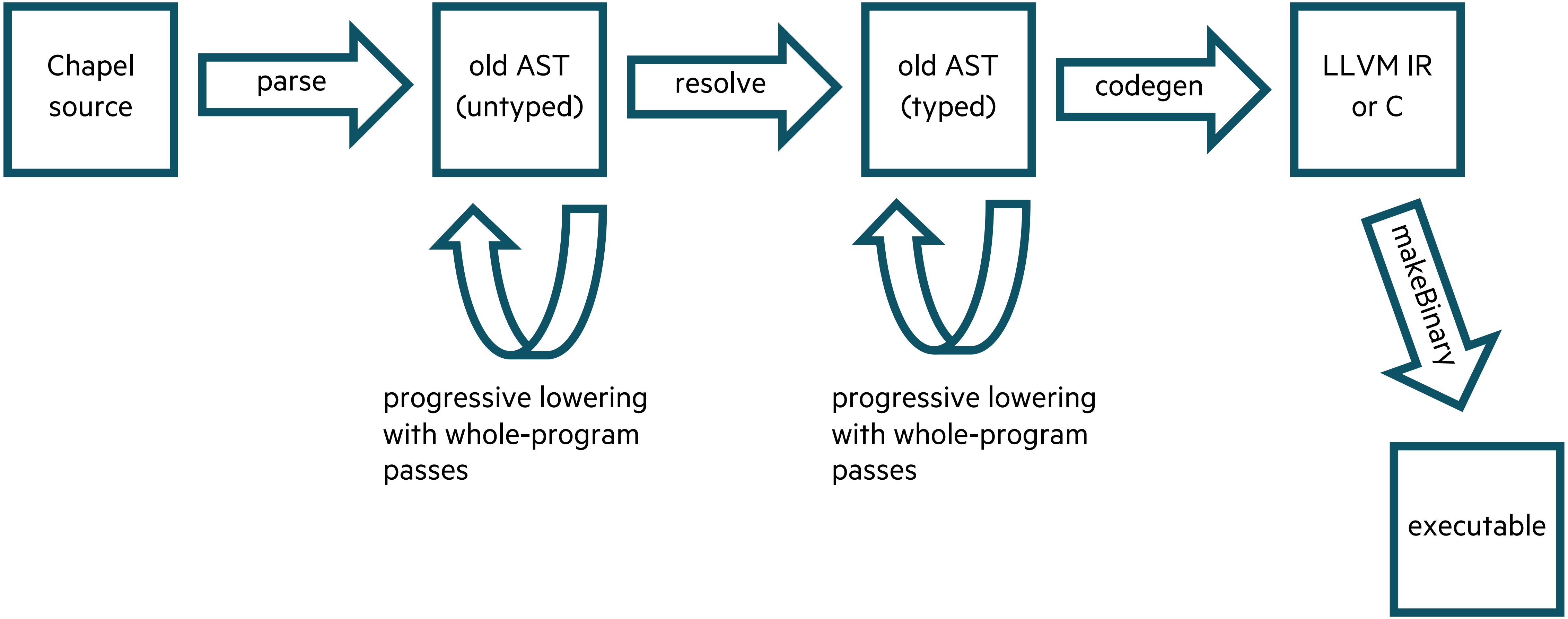
Structure and Program Representation

- The compiler is structured only for whole-program analysis, preventing separate/incremental compilation
- Unclear how to integrate an interpreter, provide IDE support, or ‘eval’ Chapel snippets
- Compilation passes are highly coupled

Development

- The modularity of the compiler implementation needs improvement
- There is a steep learning curve to become familiar with the compiler implementation

CURRENT COMPILER IN A PICTURE



Chapel source

parse

old AST (untyped)

resolve

old AST (typed)

codegen

LLVM IR or C

makeBinary

executable

progressive lowering with whole-program passes

progressive lowering with whole-program passes

COMPILER REWORK DELIVERABLES

Incremental Compilation Frontend

- Only reparse and do type resolution on files that were edited
  - Could result in reducing compilation time
- Will still have the whole-program optimization and code-generation back-end

Separate Compilation

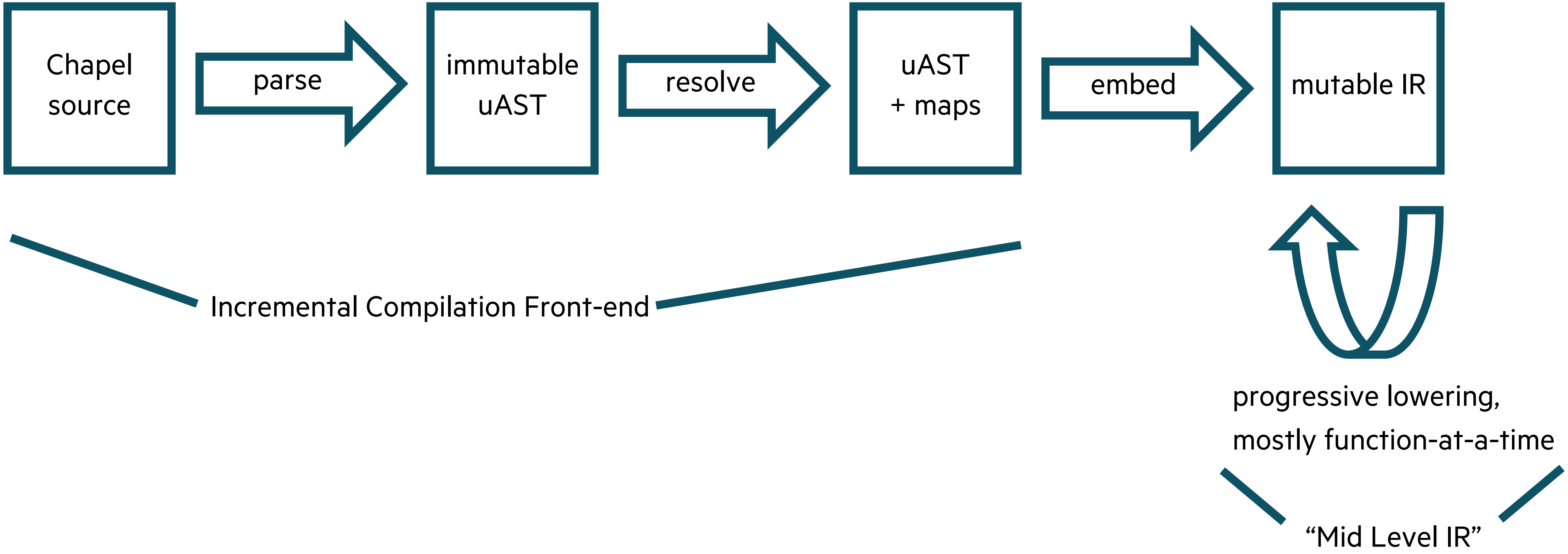
- Make most of the whole-program optimization happen per-file
- Will need a linking step for optimizations like function inlining that span files
- Should result in significantly faster compilation times

Dynamic Compilation and Evaluation

- Enable Chapel code snippets to be written and run interactively
  - e.g., in Jupyter notebooks

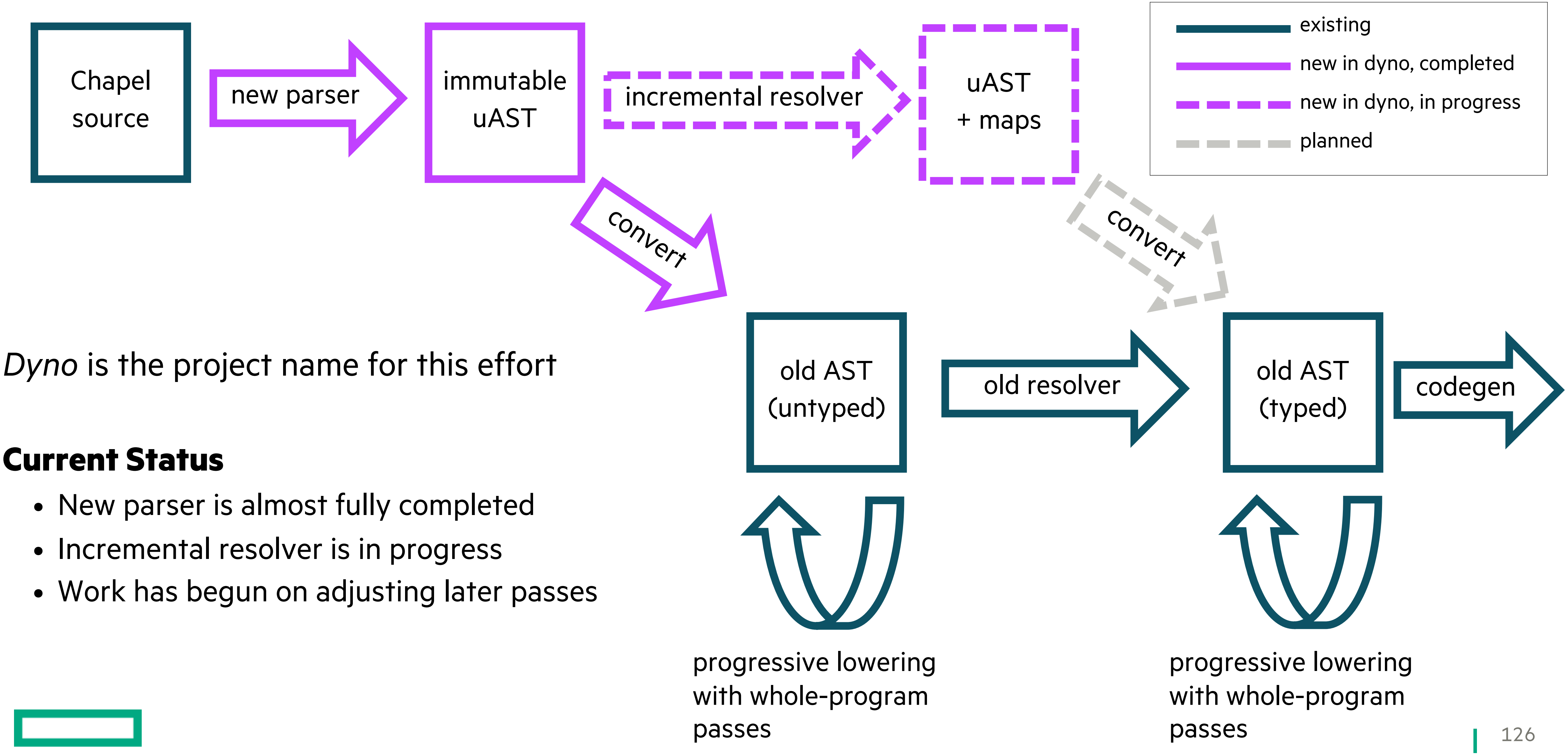
Throughout the effort, working towards improving the learning curve and error messages.

COMPILER REWORK PLAN



- The new front-end will use an untyped AST (uAST) to represent the code
- Resolution will result in maps from uAST to type & which function is called
- The old AST will initially serve as the mid-level IR

COMPILER REWORK STATUS



Chapel source

new parser

immutable uAST

incremental resolver

uAST + maps

convert

convert

old AST (untyped)

old resolver

old AST (typed)

codegen

progressive lowering with whole-program passes

progressive lowering with whole-program passes

existing

new in dyno, completed

new in dyno, in progress

planned

Dyno is the project name for this effort

Current Status

- New parser is almost fully completed
- Incremental resolver is in progress
- Work has begun on adjusting later passes

OTHER COMPILATION SPEED EFFORTS

- Improving compilation speed is a primary driver for the compiler rework effort
- However, that process will take some time
- In the meantime, we have made some spot improvements to the production compiler

- 1.26 included several improvements:
  - possible to opt-in to using ‘jemalloc’ for the compiler
  - possible to use the C backend with ‘--parallel-make’ and ‘--incremental’ to compile C code in parallel

SEPARATE COMPILATION

SEPARATE COMPILATION BACKGROUND

- We would like to support separate compilation
  - Challenging because there are generic functions and no equivalent to header files
  - Compiled libraries will store AST or source code for generic functions in case new instantiations are needed
- In a separate compilation scenario, both “compile” and “link” steps need a more flexible pass structure
  - “compile”: need to be able to compile a library without also re-compiling all dependencies
  - “link”: do not want to go through entire compilation process
    - rather, “link” should be limited to:
      - instantiating generics as necessary
      - connecting invocations of concrete functions to their implementations
- Neither of these are possible with the current rigid whole-program pass structure
  - Each pass is run in turn on the entire AST
  - Passes make whole-program assumptions and modify global variables

SEPARATE COMPILATION STRATEGY

Pursuing separate compilation with a 3-pronged effort:

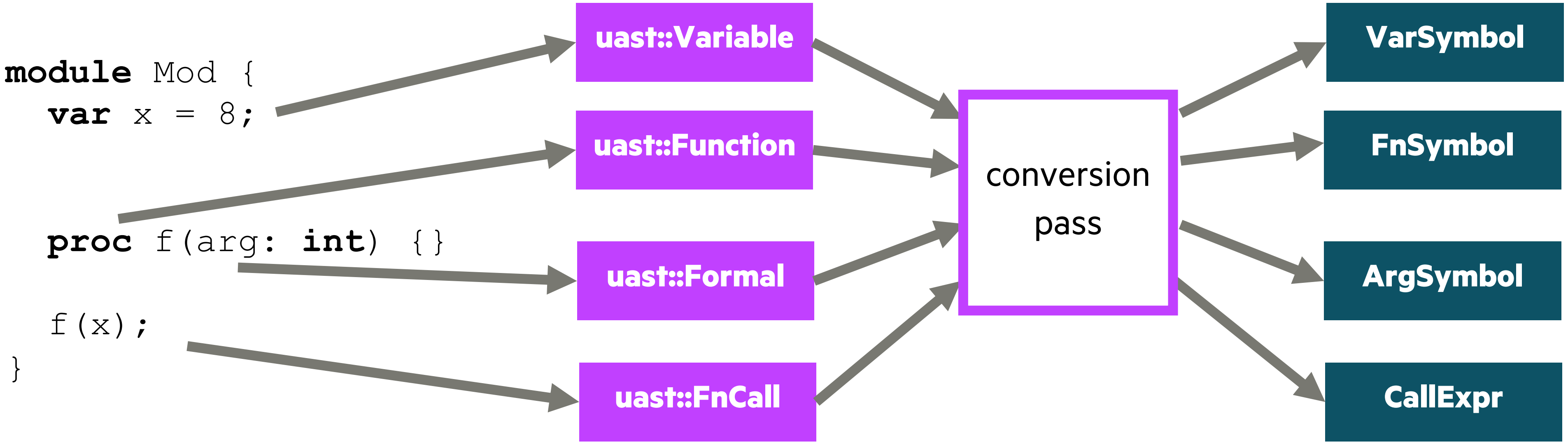
1. Rewrite the early passes of the compiler to use an incremental approach
   - parse to uAST, incrementally resolve, and convert to the old AST
2. Introduce a PassManager abstraction and adjust later passes
   - to remove whole-program compilation assumptions
3. Implement separate compilation features
   - define library file format and implement compilation and link steps

- The following sections will report on progress towards steps 1 and 2
- Step 3 has not yet been started

INCREMENTAL EARLY PASSES: PARSING TO UAST

PARSING

- Parsing is the process of reading source code and generating an abstract syntax tree
- '--dyno' flag activates a new parser that generates uAST from source code
  - uAST ("untyped AST") is more faithful to the source code than the old AST
- A new pass in the old compiler can translate uAST to the old AST
  - Most translation is done using the same helper functions that the production compiler's parser uses

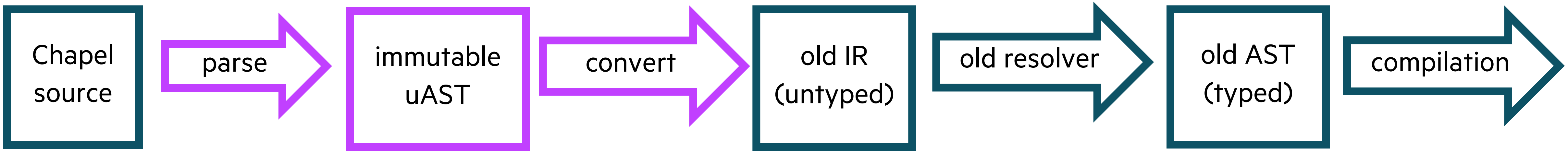


```
module Mod {
  var x = 8;


  proc f(arg: int) {}

  f(x);
}
```

PARSING: STATUS AND NEXT STEPS



- As of 1.25.0, the new parser could successfully compile:
  - Only files mentioned on the command line (i.e., not yet handling internal/standard modules)
  - All of the “Hello Worlds”
  - 9 out of 41 primers in ‘release/examples/primers’
- Now in 1.26.0:
  - New parser and uAST handle standard/internal modules
  - All primers pass with ‘--dyno’
  - 13,222/13,697 tests pass (i.e., 96%)
- In 1.27, expecting to use the new parser in production

INCREMENTAL EARLY PASSES: RESOLVING

RESOLUTION BACKGROUND

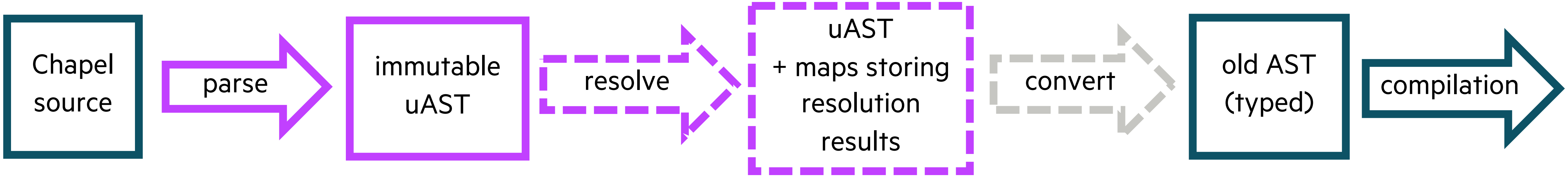
- Resolution is the process of determining what each symbol refers to and its type

```
var x: int;     // what is the type of ‘x’?
f(x);           // what does ‘x’ refer to? which ‘f’ function is called?
proc f(x) {
  writeln(x);   // which x does this refer to?
}
```

- New resolution code is implemented as incremental *queries*
  - each query has the same output when given the same input
  - queries are memoized—repeated invocations will reuse the computed result
  - queries are recomputed as needed when the input changes
  - at present, these query results are saved only in the memory of the current compiler process

INCREMENTAL RESOLVER: STATUS

- In 1.25, a prototype incremental resolver demonstrated the approach
  - scope resolution and type resolution for simple cases
- In 1.26, the incremental resolver is significantly more capable and supports:
  - type construction for tuple types and recursive types (e.g., linked lists)
  - implicit conversions and function disambiguation
  - type queries like ‘proc f(arg: ?t)’
  - generic types passed as type arguments
  - type resolution for multi-variable and tuple-style declarations and tuple expressions
- Have not yet started converter to the typed variant of old AST



INCREMENTAL RESOLVER: FEATURES

Incremental resolver is about 1/3 complete, in terms of features:

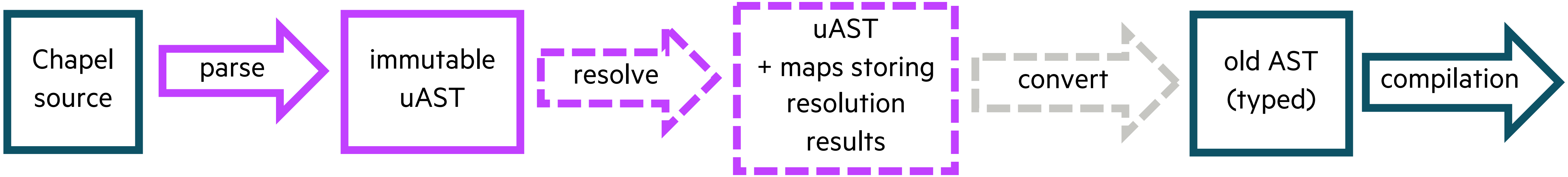
🟩	completed
□	to-do

Feature	Feature	Feature	Feature
🟩 scope resolution	🟩 resolve 'new' to initializers	□ initializers set types	□ caching of instantiations
🟩 generic instantiation	🟩 resolve '?t' in formals	□ check initializers	□ const checking
🟩 param folding	🟩 resolve tuple types	□ default functions	□ resolve try/catch
🟩 implicit conversions	🟩 type construction	□ split init	□ task/loop intents
🟩 read a file on use/import	□ varargs functions	□ copy init & copy elision	□ reflection
🟩 resolve method calls	□ casts and other operators	□ deinit	□ arrays & domains
🟩 function disambiguation	□ resolve loop index variables	□ forwarding	□ reductions

INCREMENTAL RESOLVER: NEXT STEPS

- The next steps are to:
  - adjust the '--dyno' flag to activate the incremental resolver
  - modify the downstream passes to avoid re-doing work already done by the incremental resolver
    - incremental framework computes the types & functions called; lowering transformations are still needed
  - demonstrate end-to-end compilation for some programs in this mode
    - perhaps skipping challenging functions, modules, features, or tests
  - begin to study performance of incremental resolution for a few simple use cases

- Expecting 2/3 feature completeness in incremental resolver during the next six months



LATER PASSES: PASS MANAGER

PASS MANAGER

This Effort: Developed a pass manager for progressive lowering passes

- Studied pass managers in LLVM and MLIR
- Introduced a ‘PassManager’ class
  - Runs a list of passes over a given collection of AST nodes
- Introduced a ‘Pass’ class
  - Encapsulates the functionality and state for a single pass
  - Move towards an isolated-from-above discipline
- Developed a prototype strategy for passes that update call sites
  - E.g., when inlining functions, need to modify call sites, but such scattered changes interfere with isolation goals
- Demonstrated the new framework by migrating 6 of 42 production compiler passes to it
  - The new framework is now used in the production compiler for these passes

Next Steps:

- Continue migrating passes and update the PassManager framework as needed

LATER PASSES: ADJUSTING PASSES

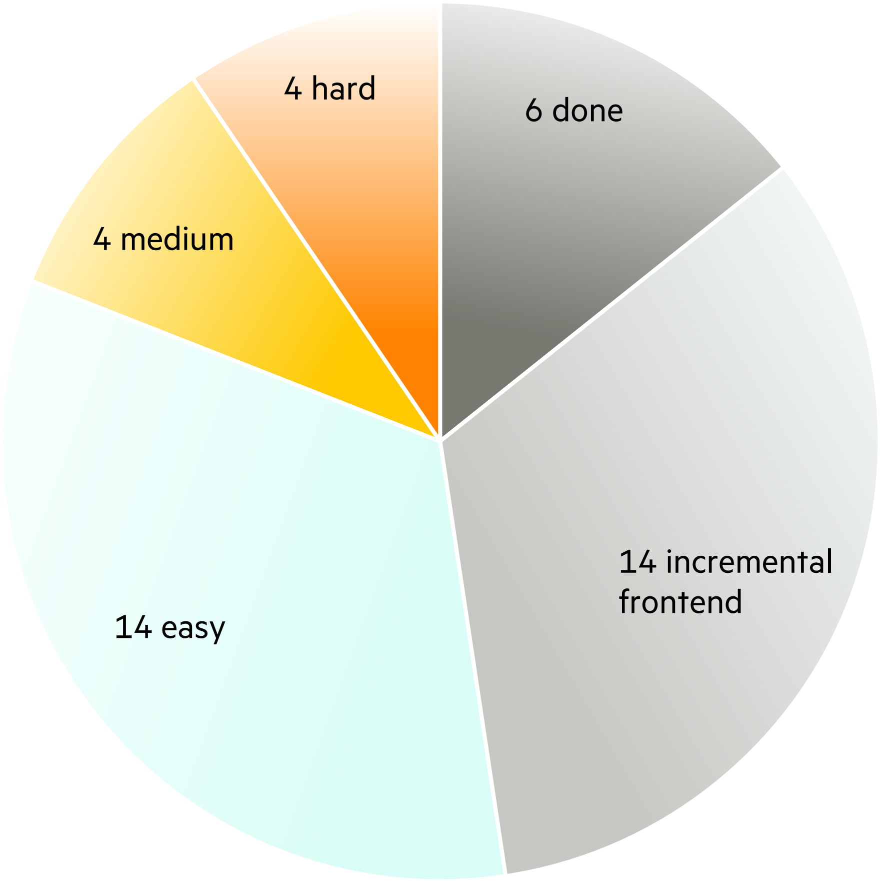
ADJUSTING PASSES

Background: Currently 42 passes in the production compiler

- ~14 passes will be handled wholly or substantially by the dyno frontend
- ~28 passes remain

This Effort:

- Prepare passes for separate compilation
- Remove whole-program compilation assumptions
- Migrate passes to a pass manager framework
  - Migrated 6 passes to the pass manager in this release
  - 22 passes remain to be addressed
  - Breakdown of passes by difficulty →



ADJUSTING PASSES: IMPACT, STATUS, NEXT STEPS

Impact: As passes are migrated, they become easier to reason about and maintain

- Groundwork for separate compilation and potential to run passes out of order
- Distinguish pass requirements from behavior

Status: Pass manager framework established; 6 passes migrated

Next Steps: Aim to migrate ~14 more passes by the 1.27 release

- Identify and separate parts of passes that should occur at “link-time” (e.g. v-table generation)
- Identify more analyses that can be moved to the incremental frontend
- Expand ‘PassManager’ framework as needed

DYNO-CHPLDOC

DYNO-CHPLDOC

Background: ‘chpldoc’ generates documentation as ‘.rst’ files by reading comments from ‘.chpl’ sources

- The generated ‘.rst’ files can then be processed by sphinx to produce HTML
- Currently, ‘chpldoc’ logic is implemented as a pass within the ‘chpl’ compiler

This Effort: Re-implement ‘chpldoc’ with the dyno framework

- Because the front-end is being reworked, the old ‘chpldoc’ implementation will no longer make sense
- Desire a ‘chpldoc’ tool using the compiler library rather than as a pass in a monolithic ‘chpl’
  - Shows the way for an ecosystem of linters and code formatting tools
- Besides the above, dyno-chpldoc can benefit from other elements of the dyno library:
  - Incremental update is now possible thanks to the new query framework
  - Leverages robust translation from uAST back to source code to make RST for function signatures

Status: Prototype ‘dyno-chpldoc’ tool has partial functionality

- passes 15/150 current chpldoc tests

Next Steps: Fix tests and replace ‘chpldoc’

LANGUAGE DESIGN IMPACT

LANGUAGE DESIGN IMPACT

- The resolver is a portion of the compiler that implements quite a lot of the language design

- The process of implementing a new resolver has brought to light many issues and questions

- These fall into 3 categories
  - problems: cases where the current language design needs some sort of adjustment to be reasonable

  - questions: cases where the compiler has to work harder than one might expect
    - leading one to wonder if those areas of the language should be simplified and easier for people to reason about

  - corner cases: cases where the language specification does not define the current behavior

LANGUAGE DESIGN ISSUES RAISED (1/2)

- Module design: problem:
  - ‘round’ enum in BigInteger conflicts with ‘round’ function in Math (#19303)
- Shadowing: problems:
  - problem: definition of shadowing is inconsistent between variables and functions (#19167)
  - problem: isMoreVisible for functions and point of instantiation (#19198)
  - problem: ‘param’ method in child class vs ‘param’ field in parent (#19474)
- Shadowing: proposal and questions:
  - proposal to simplify use/import shadowing (#19306)
  - should it be possible to shadow a symbol brought in with import? (#19160)
  - should ‘use’ statements have two shadow scopes? (#19219)
  - should it be possible to define a module that shadows an automatically included symbol? (#19312)
  - should it be possible to explicitly ‘use’/‘import’ the automatic modules? (#19313)
  - how do methods only from the type definition point interact with shadowing? (#19352)
  - should ‘use someEnum’ create a shadow scope? (#19367)

LANGUAGE DESIGN ISSUES RAISED (2/2)

- Disambiguation: question:
  - Can we simplify the function disambiguation rules? (#19195)

- Generic types and functions: questions:
  - should it be clearer when a class/record is generic? (#19120)
  - should it be clearer when a function is generic? (#19121)
  - should type constructors participate in function disambiguation? (#18817)
  - should type functions called from a function signature consult point-of-instantiation? (#19122)

- Tuple and param syntax: corner cases:
  - Should split-init be allowed for tuple declarations? (#19339)
  - How should tuple declarations interact with multiple variable declarations? (#19340)

COMPILER REWORK: SUMMARY

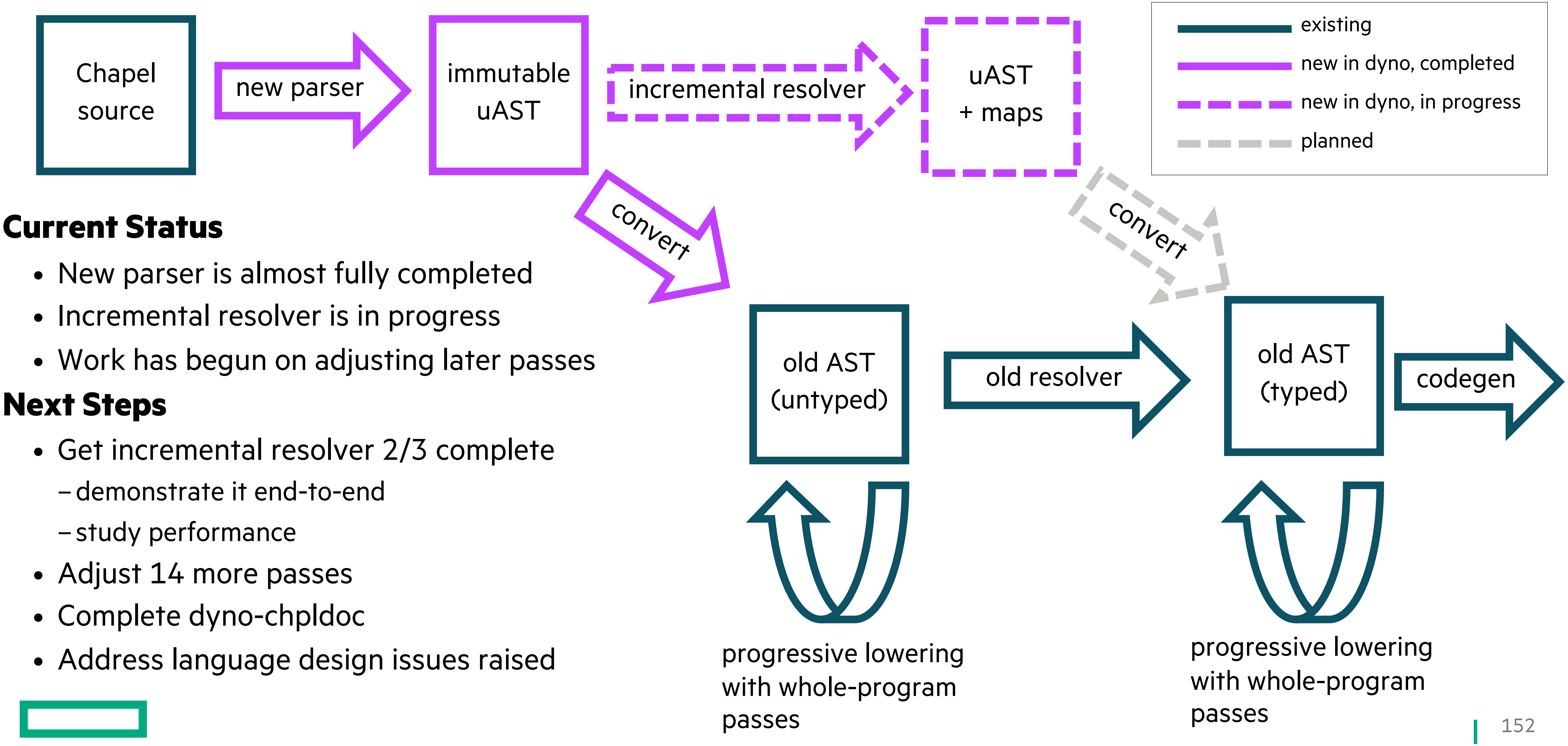
COMPILER REWORK: SUMMARY

- The current Chapel compiler has problems with:
  - speed
  - error reporting
  - structure and program representation
  - its steep learning curve

- This rework effort is addressing the speed problems through architectural adjustments
  - to enable incremental compilation and separate compilation

- During development, this rework effort is taking steps to improve the learning curve
  - more modular design
  - generated API documentation

COMPILER REWORK: STATUS



Current Status

- New parser is almost fully completed
- Incremental resolver is in progress
- Work has begun on adjusting later passes

Next Steps

- Get incremental resolver 2/3 complete
  - demonstrate it end-to-end
  - study performance
- Adjust 14 more passes
- Complete dyno-chpldoc
- Address language design issues raised

THANK YOU

https://chapel-lang.org
@ChapelLanguage