Compiler and Tool Improvements

Chapel version 1.20

September 19, 2019

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

Outline

- Creating and Using Libraries
- Mason Improvements
- LLVM Back-end Improvements

Creating and Using Chapel Libraries

Chapel Libraries: Background

- Have had a draft capability to create Chapel libraries
  - '--library' flag generates library instead of executable
    - 'main()' function not valid in this compilation mode
  - Supported single-locale libraries for C, Python, and Fortran
    - But not multi-locale libraries
  - Supported primitive types and arrays (some natively, some opaquely)
  - Much work remained

Multi-locale Libraries

Multi-locale Libraries: Background

- Had support for single-locale libraries and multi-locale executables
  - But not multi-locale libraries

- Chapel expects to control how a program is distributed
  - Trickier when Chapel doesn't own 'main()'

- By last release, had determined strategy for support but not yet implemented it

Multi-locale Libraries: This Effort

- Added multi-locale '--library' support
  - Like non-library multi-locale, users don't have to worry about launching
    - Agnostic to user program running on compute or login node
  - Similar behavior for routine calls, argument types, etc. to single-locale
  - Requires ZMQ installation

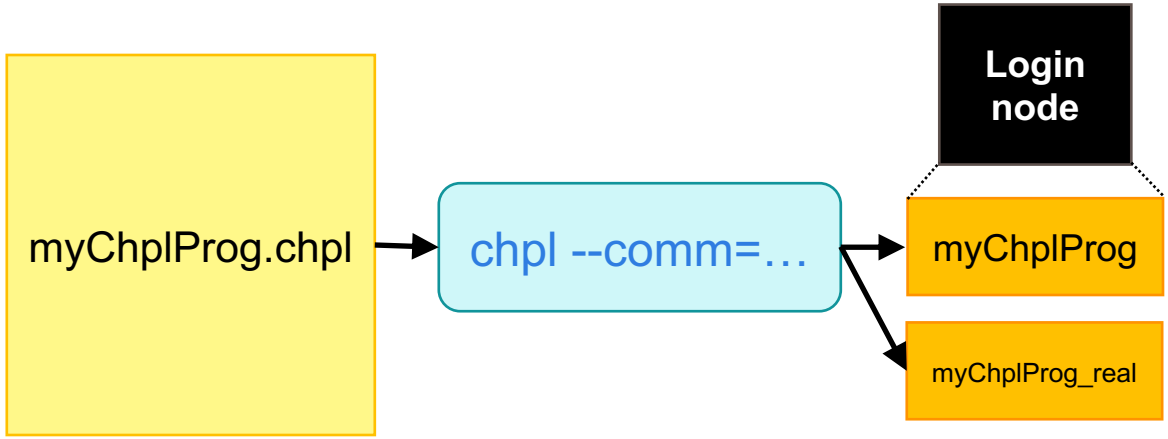
Multi-locale Libraries: Example Program

```
// foo.chpl
export proc hello() {
  coforall loc in Locales do on loc {
    writeln("Hello from locale ", loc);
  }
}
```

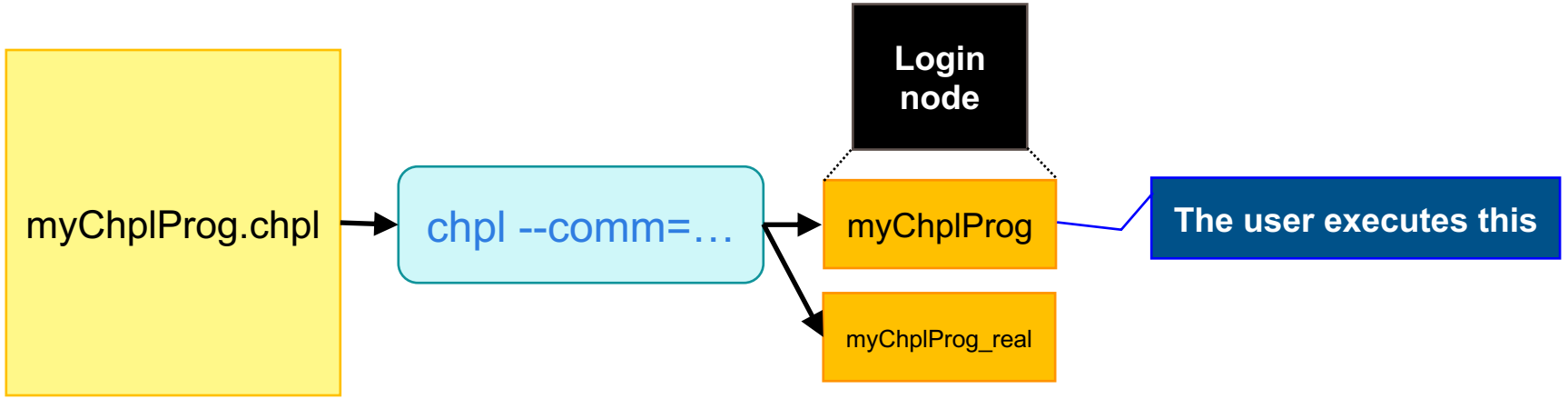
```
// use_foo.py
import foo
// specify number of locales
foo.chpl_setup(2)
foo.hello()
foo.chpl_cleanup()
```

```
$ chpl --library-python --comm=gasnet foo.chpl
$ export PYTHONPATH=lib/
$ python3 use_foo.py
Hello from locale 1
Hello from locale 0
```

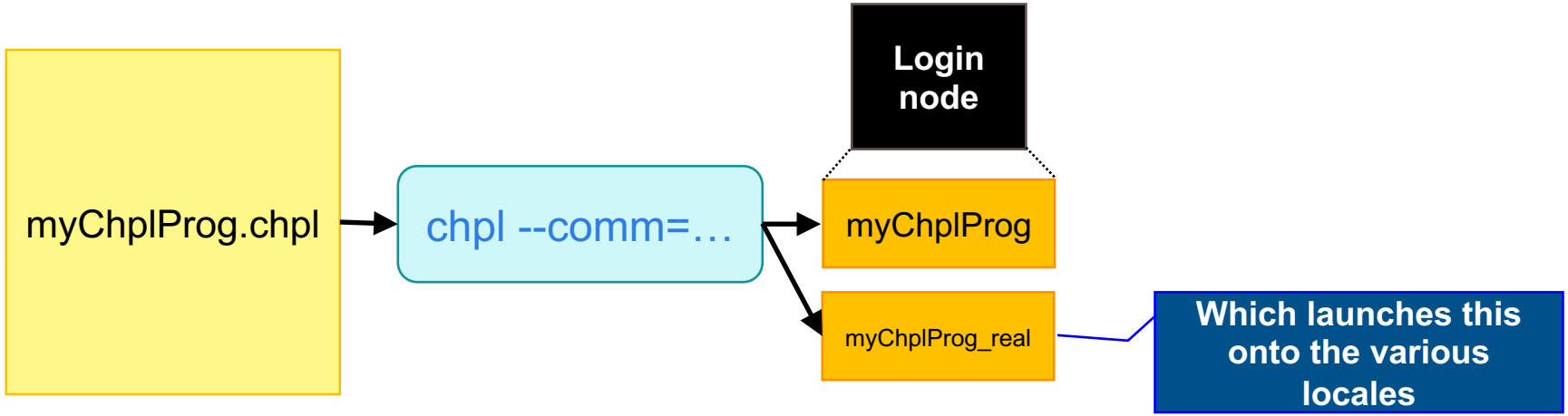

Typical Multi-Locale Compilation + Execution



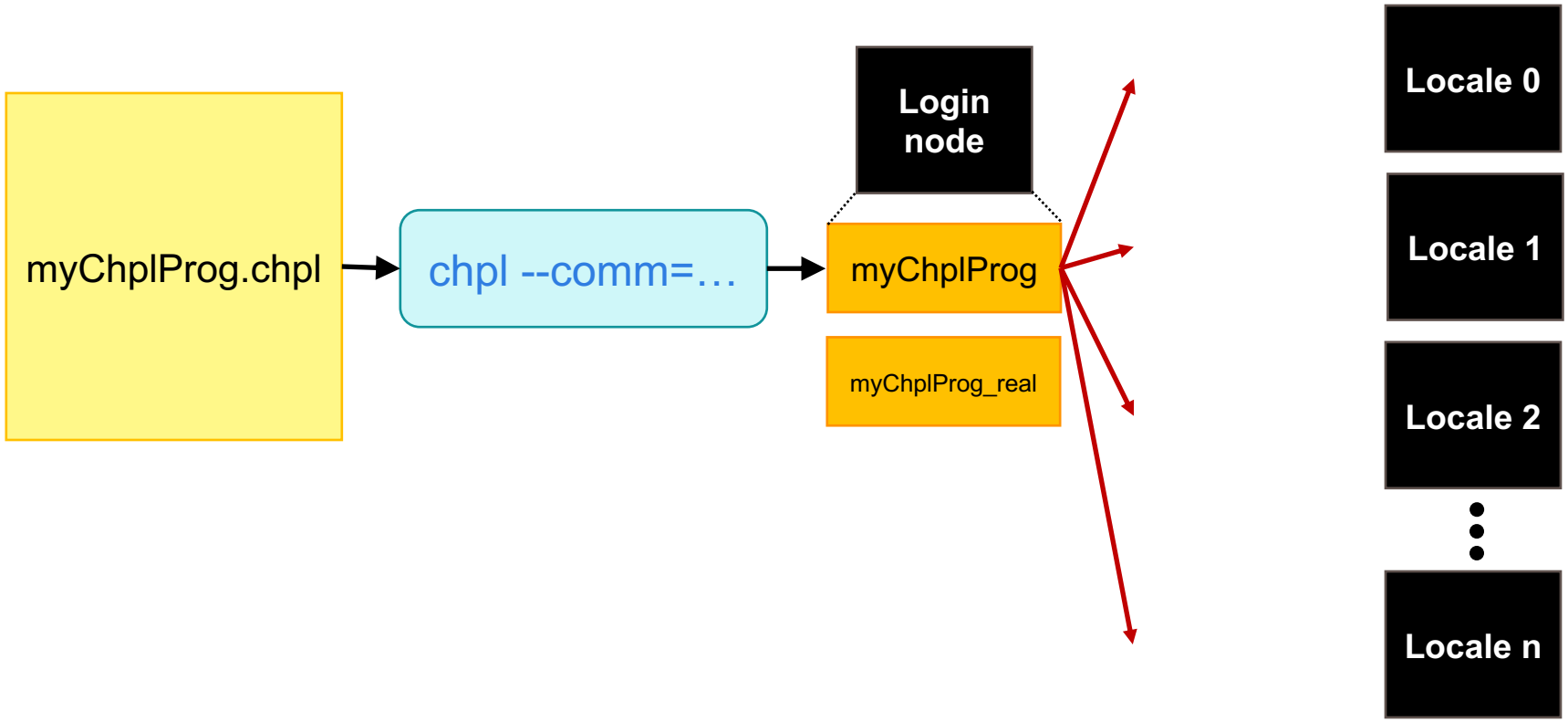
Typical Multi-Locale Compilation + Execution



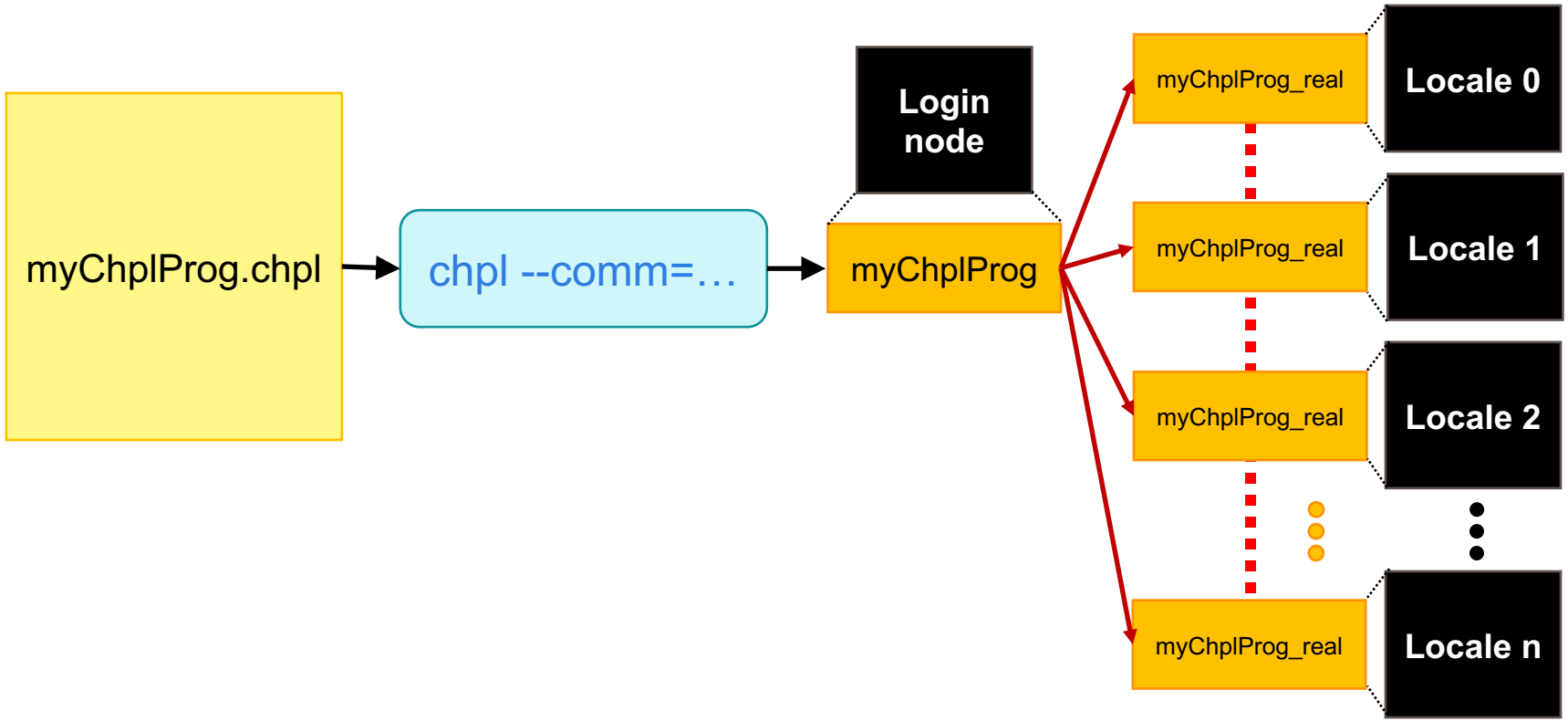
Typical Multi-Locale Compilation + Execution



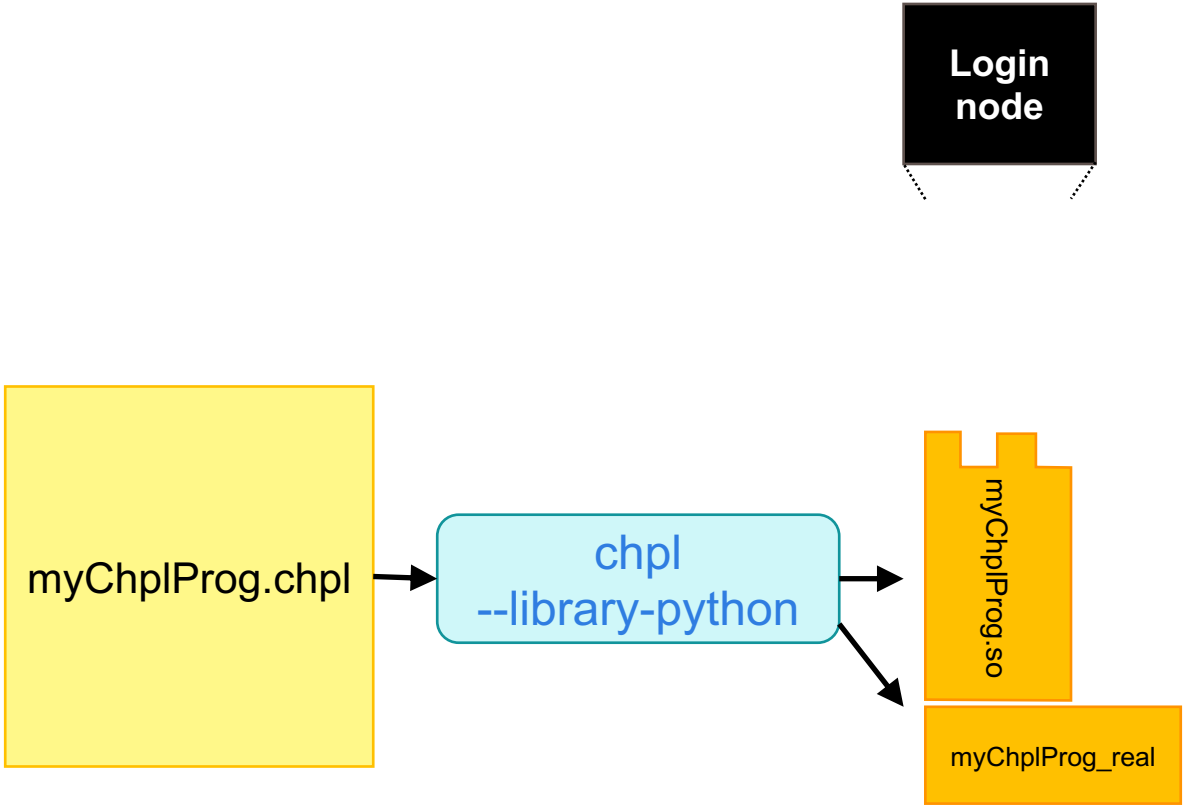
Typical Multi-Locale Compilation + Execution



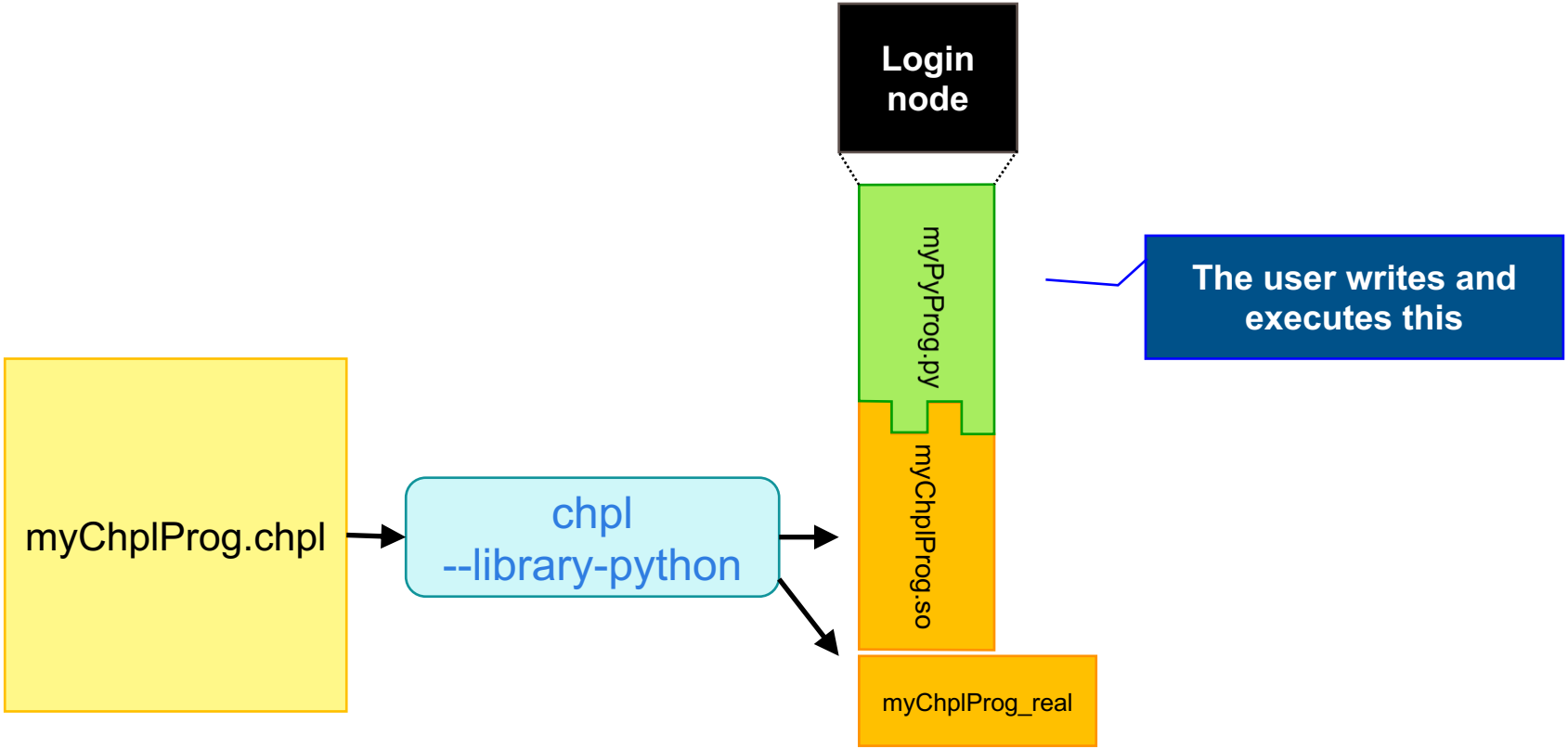
Typical Multi-Locale Compilation + Execution



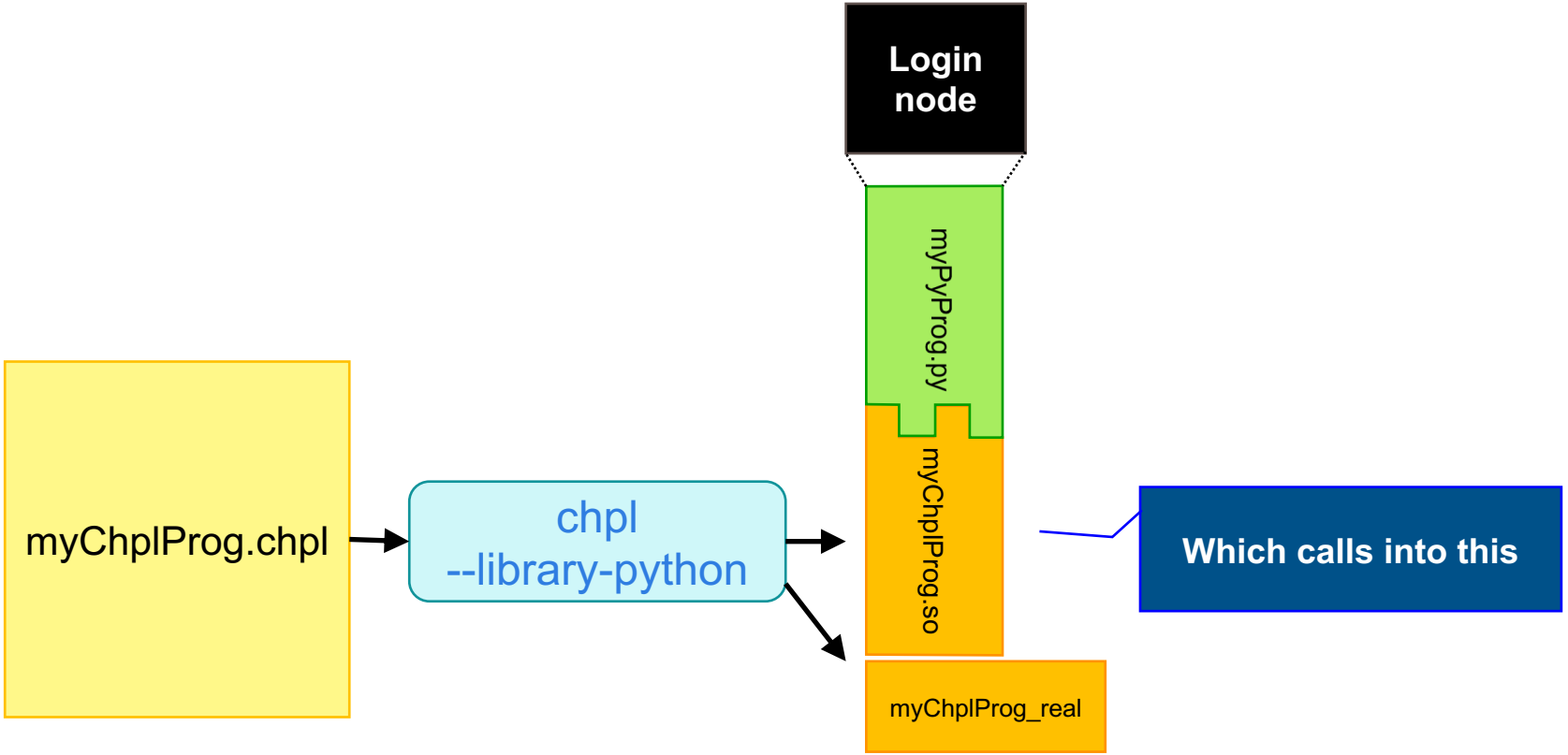
Multi-Locale Chapel-Python Interop



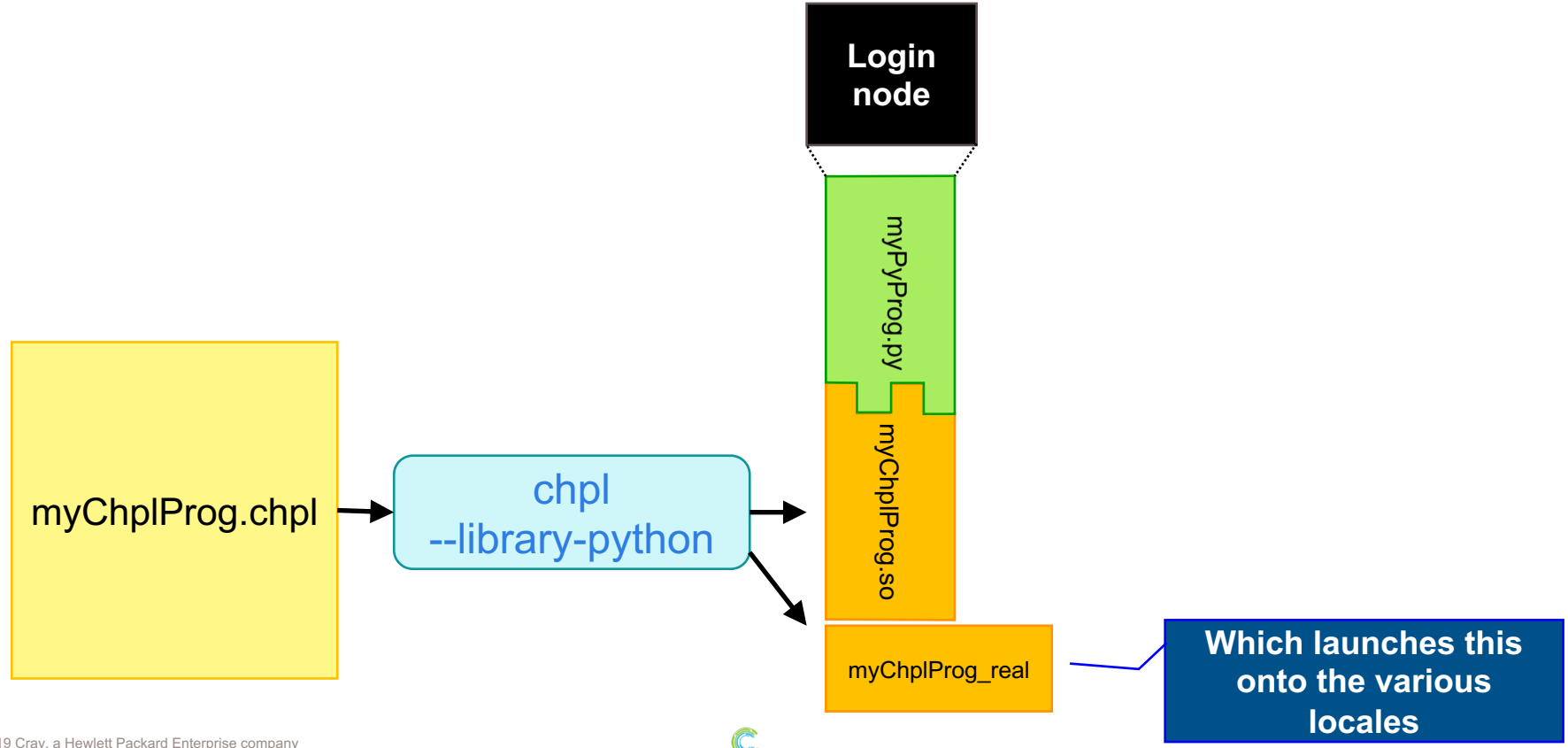
Multi-Locale Chapel-Python Interop



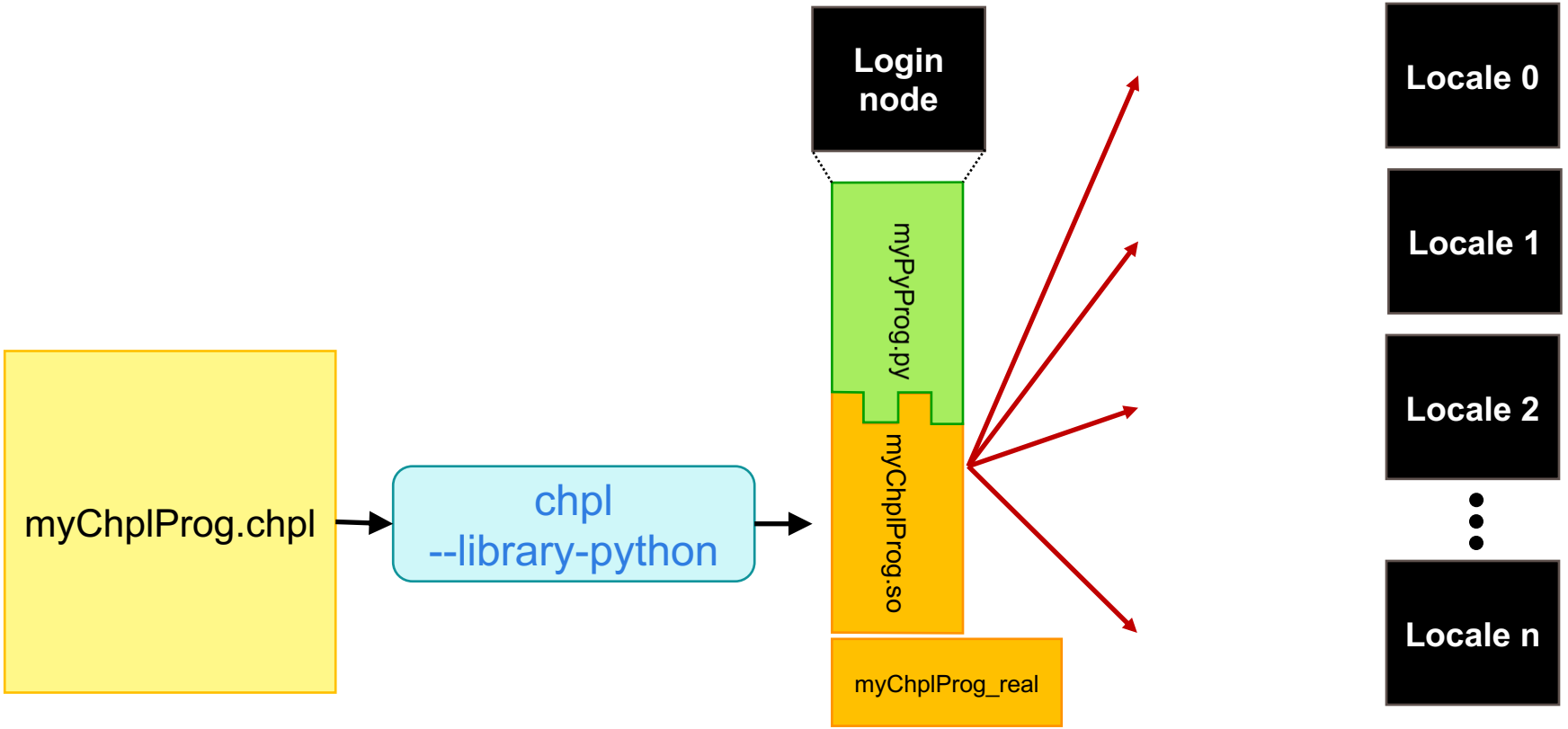
Multi-Locale Chapel-Python Interop



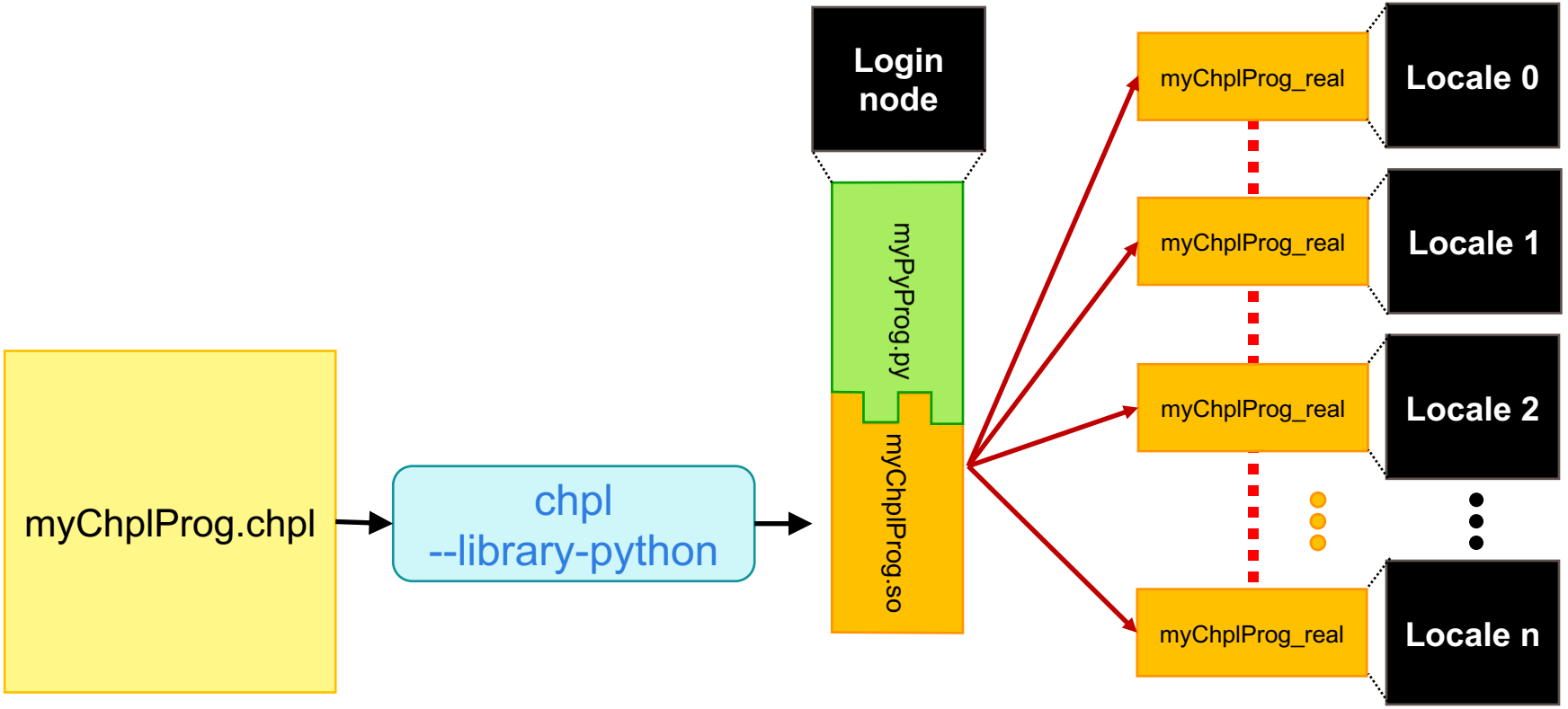
Multi-Locale Chapel-Python Interop



Multi-Locale Chapel-Python Interop



Multi-Locale Chapel-Python Interop



Multi-locale Libraries: This Effort

- Library initialization now requires specifying the number of locales
  - In multi-locale with Python, ‘chpl_setup()’ now takes an argument

```
chpl_setup(); // call to set up library for single-locale Python (unchanged)
chpl_setup(numLocs); // call for multi-locale Python (new arg!)
```

  - With C, pass ‘-nl <num>’ in addition to other arguments to ‘chpl_library_init()’

```
// One strategy for multi-locale C:
int argChapelC = 3;
char* argChapelV[3] = {argv[0], "-nl", "2"};
chpl_library_init(argChapelC, argChapelV);
```


Multi-locale Libraries: Status

- Supports most primitive types and Chapel strings
  - Strings only support ‘[const] in’ intent due to copying contents
  - Doesn’t support ‘complex’ or arrays yet

- Supports Python and C
  - Doesn’t support Fortran just yet
  - Doesn’t support LLVM back-end

String Support

String Support: Background

- Exported Chapel routines did not support exporting the ‘string’ type at all:
  - Instead, users had to use ‘c_string’

```
// Wouldn’t work, strings can’t be in exported routine declarations
export proc foo(s: string): string {
      return s + ", yo";
}
```

- This presented some problems:
  - Strings are a fundamental type
    - e.g., Arkouda’s RPC protocol is implemented in terms of strings
  - We want to deprecate support for ‘c_string’, not put more weight on it

String Support: This Effort

- We added support for exporting strings to C and Python
- When exporting to C, a Chapel routine's formal arguments map to 'c_string'
  - Returned strings map to 'char*'

```
export proc useful(in s: string): string { … }

/* Making use of this routine in C... */
char* msg = useful("hello");
printf("%s\n", msg);
chpl_free(msg);
```

- When exporting to Python, Chapel strings map to the Python 'bytes' type

String Support: Supported Intents

- Single- and multi-locale libraries support different formal argument intents
- In multi-locale and Python libraries:
  - ‘string’ arguments must have the ‘in’ intent
  - strings may only be returned by value

```
export proc foo(in s: string): string;
```

- Restrictions on argument intents exist for multi-locale libraries by design
  - Arguments are copied when they are communicated to the multi-locale library
- For single-locale Python, these restrictions may be temporary

String Support: Impact, Next Steps

Impact:

- Strings can now be passed between Chapel and client languages

Next Steps:

- Support more formal argument intents in both single- and multi-locale libraries
- Support returning by ref intent in single-locale libraries
- Use Python ‘string’ rather than ‘bytes’ to interoperate with Chapel ‘string’
- Support the Chapel ‘bytes’ type
- Minimize copying via compiler support

Libraries and LLVM

Libraries and LLVM: Background, This Effort

Background: '--library --llvm' created libraries usable from C programs

- This mode was unable to create libraries for Python
- Several '--library-*' flags were not supported

This Effort: Extended support to include:

- '--dynamic', enabling the creation of shared libraries
- '--library-python', enabling Python interoperability
- '--library-makefile', generating Makefile to make compiling with the lib easier
- 'CHPL_LIB_PIC', enabling position independent code

Chapel Libraries: Impact and Next Steps

Chapel Libraries: Impact

- Multi-locale and Chapel string support make interoperability stronger

- Chapel can now support user scenarios like Arkouda's
  - Almost all of Arkouda can be compiled as a multi-locale library
    - Remaining issues relate to null bytes embedded in strings
    - Support for 'bytes' arguments in exported routines should help

Chapel Libraries: Next Steps (short-term)

- Support other intents for string formals in exported routines
  - For single-locale libraries, we want to support all intents in the future
  - For multi-locale libraries, we want extend support to include ‘inout’

- Support passing ‘bytes’ between Python and Chapel
  - This should permit all of Arkouda to be compiled as a multi-locale library

Chapel Libraries: Next Steps (medium-term)

- Extensions for LLVM compiler back-end
  - Support multi-locale and Fortran libraries with LLVM
    - multi-locale libraries need adjustments to their build process for LLVM
  - Fix ABI incompatibility when returning arrays

- Support exporting a wider variety of types and symbols
- More robust default values for Python

Chapel Libraries: Next Steps (longer-term)

- Enable users to link against multiple Chapel libraries
- Enable Chapel libraries to be used by Chapel programs
- Adopt a long-term serialization strategy and RPC protocol for multi-locale libraries
  - Currently, the compiler generates serialization code
  - Our RPC protocol is simple and blocking
  - Protobuf and Google RPC are viable alternatives
- Extend interoperation to other languages
  - C++ interoperability has been requested
- Support libraries built with '--no-local' and 'CHPL_LAUNCHER != none'
  - Using similar strategy to multilocale libraries

Mason Improvements

Mason: Background

- Mason is a package manager for Chapel programs
  - Introduced with Chapel 1.16.0
- Mason is still under development and has some known shortcomings:
  - Preventing web queries requires throwing '--no-update' with most commands
    - This can become tedious when working in an offline environment
  - Publishing mason packages requires many manual steps
    - This increases the barriers for users to publishing their packages
  - The mason test system is very simple, only based on exit code
    - This puts a lot of work on the user and lacks flexibility

Mason: This Effort - Using Mason Offline

- New ‘MASON_OFFLINE’ environment variable indicates user is working offline
  - Sets ‘--no-update’ flag by default for relevant mason commands
    - Can be overridden by throwing ‘--update’
  - Disables usage of ‘mason external’ commands that require internet access
    - Specifically, commands that install Spack or Spack packages
  - Does not prohibit accessing local registries on the filesystem

Mason: This Effort - Mason Publish

- New ‘mason publish’ command helps users publish packages

```
mason publish [options] <registry>
```

- Automates creation of version tag, changes to registry, and pushing to remote
- Supports local and remote registries, defaulting to chapel-lang/mason-registry
- Some prerequisites still remain for remote registries:
  - User must host package on a remote git repository
  - User must fork registry to which they are publishing
- Supports flags to help guide users:
  - ‘--dry-run’ flag checks if general package prerequisites are met
  - ‘--check’ flag checks if prerequisites are met for a specific registry

Mason: This Effort - Mason Publish Example

```
> mason publish
...
remote:  https://github.com/ChapelUser/mason-registry/pull/new/Foo
remote:
To github.com:ChapelUser/mason-registry
 * [new branch]      Foo -> Foo
------------------------------------------------------------------
Go to the above link to open up a Pull Request to the mason-registry

> mason publish /path/to/local/registry
Successfully published package to /path/to/local/registry
```


Mason: This Effort - Mason Test

- ‘mason test’ serves as the test runner for the new UnitTest module
  - As a result, ‘mason test’ can be used outside of a mason package

```
mason test [options] <path>
```

- ‘mason test’ remains backwards-compatible with exit-code-based testing
  - If UnitTest module not used, entire file is considered a single test
    - Exit code of 0 is required for pass
- Mason packages can contain a mix of exit-code and UnitTest tests

Mason: Status, Next Steps

Status:

- Offline experience for mason improved
- Publishing mason packages to registries is easier
- Mason packages can now include tests written with the new UnitTest module

Next Steps:

- Continue improving mason:
  - CI testing of mason-registry
  - Centralized package cache
  - Package security

Improvements to the LLVM Back-end

LLVM: Background

- LLVM is a compiler optimization framework
  - Actively developed and constantly improving
- The Chapel compiler generates C code by default
  - Runs a C compiler to compile the generated code
  - But can generate LLVM Intermediate Representation instead
- We want the Chapel compiler to use LLVM by default
  - To reduce maintenance
    - supporting a variety of C compilers requires significant effort
  - To enable new optimization opportunities

LLVM: This Effort

- Upgraded the bundled LLVM to version 8 and fixed a variety of bugs
- A Google Summer of Code project made improvements to --llvm:

Student	Mentors
Mohammed Nafees	Michael Ferguson Przemek Leśniak

  - The generated llvm.ident now includes Chapel version information
    - llvm.ident identifies the compiler that generated the LLVM IR
  - Stopped generating C in most cases when using --llvm
  - Emit llvm.lifetime.start / llvm.lifetime.end metadata to aid optimization
  - Improved --llvm -g debug information generation

LLVM: Next Steps

- Implement full ABI compatibility
- Make --llvm the default for the 1.21 release
- Study more benchmarks to try and improve performance
- Continue to work on Region Vectorizer integration
  - Mark more loops as vectorizable
- Improve alias analysis between C and Chapel types
- Improve debugging experience with --llvm -g

For More Information

For a more complete list of compiler and tools changes in the 1.20 release, refer to 'New Tools / Tool Changes', 'Interoperability Improvements', 'Portability', 'Compiler Improvements', and other sections in the CHANGES.md file.

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.

THANK YOU

QUESTIONS?


chapel_info@cray.com


@ChapelLanguage


chapel-lang.org

cray.com


@cray_inc


linkedin.com/company/cray-inc-/