Portability and Interoperability Improvements

Chapel version 1.19

March 21, 2019

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

Outline

- 'ofi' Communication Layer
- Chapel Libraries
  - Exporting Arrays
  - Interoperating with Fortran
  - Chapel in Python Modules
  - LLVM For Libraries
  - PIC Library Code
  - Multilocale Libraries
  - Next Steps for Libraries
- Interoperating with C Arrays
- Renaming Extern Variables

© 2019 Cray Inc.

'ofi' Communication Layer

© 2019 Cray Inc.

ofi Comm Layer: Background and Progress

Aim: a single comm layer for HPC networks, with:

- minimal software overhead
- good day-1 performance for new networks

Last release: ofi “mock-up” standalone multi-node proxy for comm layer activities

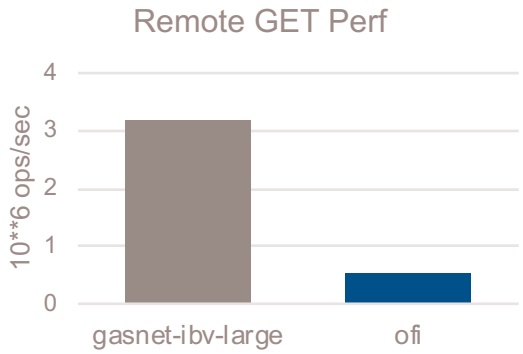
This release: functional comm=ofi implementation

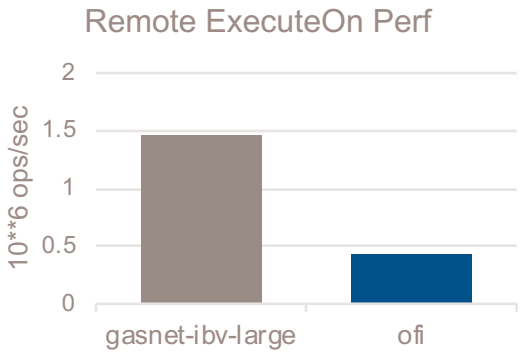
- Libfabric has *providers* for each network, similar to GASNet conduits
  - Provider capabilities may vary; user (comm layer) must adapt or avoid
- All but 2 tests pass with 'sockets' and 'verbs' providers on InfiniBand cluster
- All examples and most multilocale tests pass with 'gni' provider on Cray XC
  - Failures in multilocale were Cray XC-specific, not libfabric-specific

© 2019 Cray Inc.

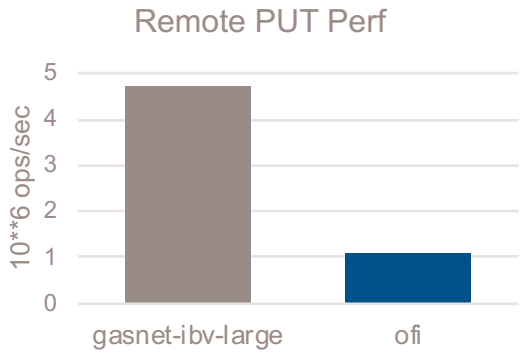
ofi Comm Layer: Status

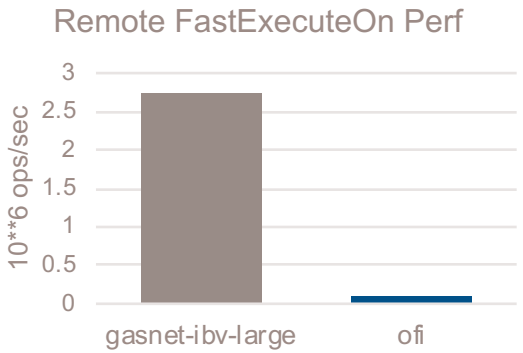
- InfiniBand results:





- Microbenchmark performance with default settings is not yet on par with comm=gasnet



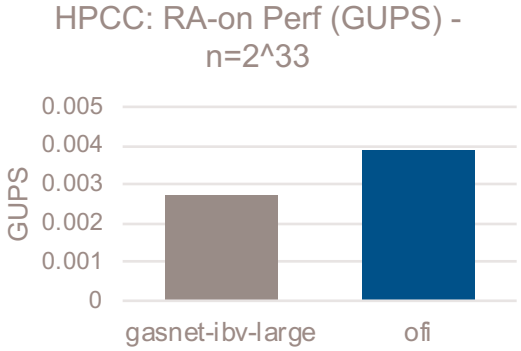


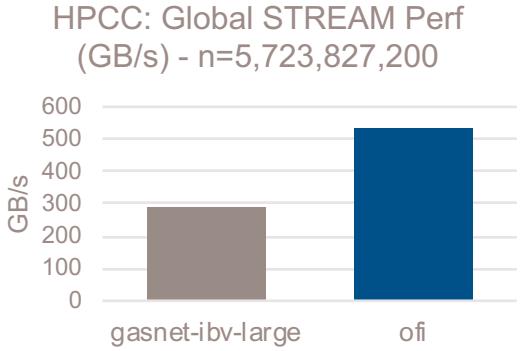
© 2019 Cray Inc.

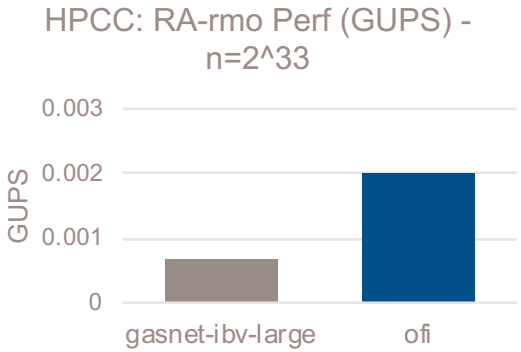
ofi Comm Layer: Status

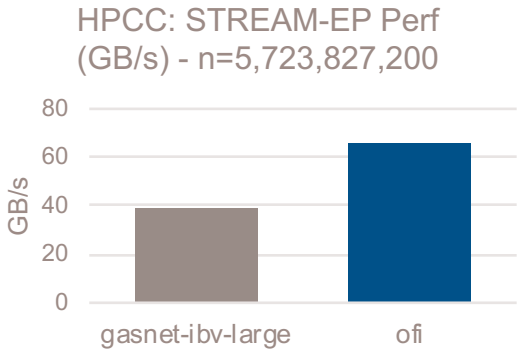
- InfiniBand results:

- Poor microbenchmark performance doesn't necessarily mean poor performance overall





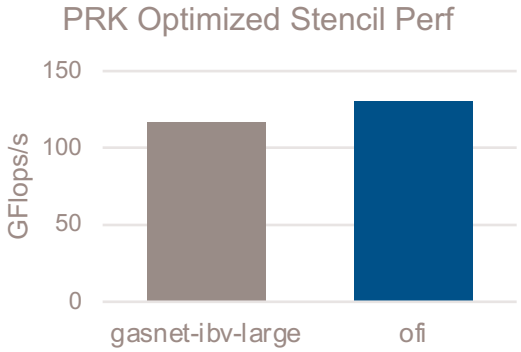




© 2019 Cray Inc.

ofi Comm Layer: Status

- InfiniBand results:



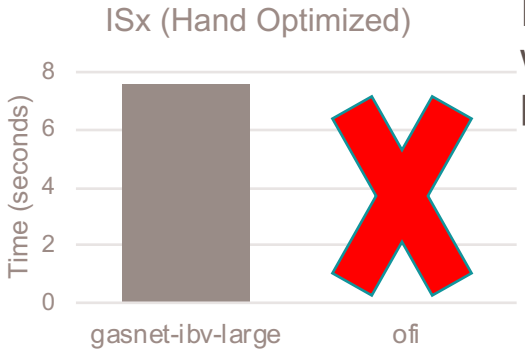
- Poor microbenchmark performance doesn't necessarily mean poor performance overall

© 2019 Cray Inc.

ofi Comm Layer: Status

- InfiniBand results:

- There are still some anomalies …



With comm=ofi, ISx got an InfiniBand-related error when sized for multilocale performance testing.

© 2019 Cray Inc.

ofi Comm Layer: Next Steps

- Complete functionality
  - Fix bugs (ISx, e.g.) revealed by preliminary performance testing
  - Add support for nonblocking GET/PUT and unordered atomics
  - Integrate better with Chapel launchers
- Begin nightly functional and performance testing
- Tune for performance
  - Libfabric interactions (examples: provider selection, progress threads)
  - Provider/platform optimizations (examples: 'verbs'-specific settings, Shasta)

© 2019 Cray Inc.

Creating and Using Chapel Libraries

© 2019 Cray Inc.

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
  - The 1.18 release added:
    - Support for 1D rectangular array arguments and return types
    - Many helper flags, especially for use with C client programs
    - The ability to compile into a Python module
  - Much work remained

© 2019 Cray Inc.

Exporting Arrays

© 2019 Cray Inc.

Exporting Arrays: Background

- Last release added support for 1D rectangular array arguments and return types
  - Available for both Python and C interoperability
    - Translated to NumPy array in Python, 'chpl_external_array' struct in C

- Multidimensional rectangular arrays and other array types not supported

© 2019 Cray Inc.

Exporting Arrays: Background

- Motivation: want to tell Chapel to manipulate array that C/Python can't represent
  - Stop-gap solution until we can natively support them

ChapelLib.chpl:

```
const D = {1..n} dmapped Block(…);
export proc getArr(): [D] int {…}
export proc manipulate(x: [D] int) {…}
```

PythonProg.py:

```
import ChapelLib
x = ChapelLib.getArr() // details of x not representable just yet
ChapelLib.manipulate(x)
```

© 2019 Cray Inc.

Exporting Arrays: This Effort

- Added opaque representation of most other array types
  - 'chpl_opaque_array' struct for C, 'ChplOpaqueArray' class for Python
  - Allows us to pass unrepresentable arrays around
  - Can only be obtained from an exported Chapel function that returns an array
    - Unlike 'chpl_external_array', which can be created in C
    - Only array types unsupported by 'chpl_external_array' will generate it

© 2019 Cray Inc.

Exporting Arrays: This Effort

- Contents cannot be accessed in C/Python
  - Can only send as argument to Chapel functions

```
const D = {1..n} dmapped Block(…);
export proc foo(x: [D] int) {…}
// transforms into this declaration in C
void foo(x: chpl_opaque_array) {…}
```

© 2019 Cray Inc.

Exporting Arrays: Next Steps

- Support interoperation between multidimensional Chapel and NumPy arrays
  - Would support these arrays more natively than by using opaque arrays
  - Intend to take approach similar to Fortran array support (see next section)

- Interoperate on arrays in place
  - Currently performs a copy when crossing Python-Chapel function borders

© 2019 Cray Inc.

Interoperating with Fortran

© 2019 Cray Inc.

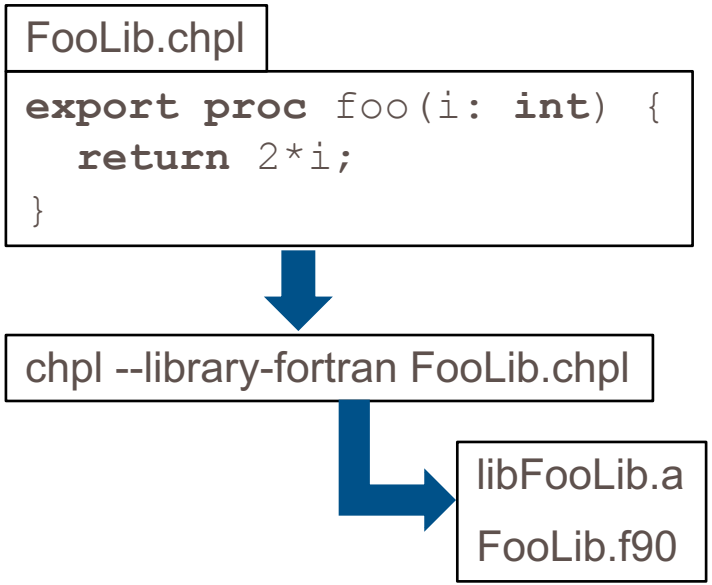
Fortran Interoperation: Background

- Fortran is popular in the HPC and scientific programming space
- Chapel and Fortran can both interoperate with C
- Seemed possible to use the C bindings of both languages to tie them together

© 2019 Cray Inc.

Fortran Interoperation: This Effort (Fortran→Chapel)

- Added support for calling Chapel functions from Fortran
  - Using the compiler flag '--library-fortran' builds a library with 'export' functions
    - Also generates a Fortran interface module for the 'export' functions



© 2019 Cray Inc.

Fortran Interoperation: This Effort (Fortran→Chapel)

- Fortran array arguments are handled specially by the Chapel compiler
  - Declare the 'export' function's argument with regular Chapel array syntax
  - Compiler translates from Fortran array to regular Chapel array
    - Uses ISO/IEC specified structures for Fortran/C interoperability
    - Can then be iterated over in parallel with 'forall', sliced, reduced, etc.

```
export proc foo(A: [] real) {
  forall i in A.domain do
    A[i] = i*10;
}
```

```
program PassArray
  use FooLib ! chpl-generated interface
  real(kind=8), dimension (9) :: arr
  call foo(arr)
end program PassArray
```

© 2019 Cray Inc.

Fortran Interoperation: This Effort (Chapel→Fortran)

- Added support for calling Fortran functions from Chapel
  - Separately compile the Fortran code with a Fortran compiler
  - Declare the interface using marked 'extern' functions in Chapel
  - Chapel compiler generates the signature of the marked 'extern' functions
    - Currently marked using 'pragma "generate signature"'
    - Fills the purpose of the header files used in C 'extern' support
  - Chapel compiler links the Fortran binary with the Chapel binary
    - Requires explicitly linking with the Fortran runtime library

```
$ chpl chplCode.chpl ftnCode.o -L$LIBGFORTRAN_PATH -lgfortran
```

© 2019 Cray Inc.

Fortran Interoperation: Impact, Next Steps

Impact: Chapel and Fortran functions can call each other

- Can use primitive types as arguments and return values
- Compiler automatically generates Fortran interface code
- 1D Fortran arrays automatically convert to regular Chapel arrays
  - Enables parallel iteration, slicing, reductions, etc.

Next Steps: Continue to improve support

- Support more complex arguments and return types
  - Multidimensional arrays, records, tuples
- Support multilocale Chapel as a library for Fortran

© 2019 Cray Inc.

Chapel in Python Modules

© 2019 Cray Inc.

Chapel Python Modules: Background

- Can compile Chapel libraries into a Python module
  - Imported in Python code like any other module

```
import ChapelLib; // for some ChapelLib.chpl
```

  - Supports primitive types and array arguments
    - Arrays are returned as NumPy arrays (but lists accepted as arguments)

© 2019 Cray Inc.

Chapel Python Modules: This Effort

- Added support for literal argument default values

```
// can call both 'foo' and 'bar' from Python w/ or w/o the 'x' arg
export proc foo(x: int = 4) {…}
export proc bar(x: c_string = "blah") {…}
```

  - Complicated default values will be dropped on the floor, with a warning
    - Must always provide that argument when calling the function from Python

```
export proc foo(x: int, y: int = x) {…}    // must call with ‘x’ and ‘y’
export proc bar(x: int = someGlobalVar) {…} // must call with 'x' arg
export proc baz(x: complex = 4 + 3i) {…}    // must call with 'x' arg
```

© 2019 Cray Inc.

Chapel Python Modules: This Effort

- 'c_ptr' Chapel arguments now supported in Python code
  - With NumPy arrays:

```
arr = numpy.empty(6, dtype=numpy.double)
ChapelLib.func(arr) // usually want to specify size of array as another arg
```

  - With ctypes pointers:

```
dub = ctypes.c_double()
doubPtr = ctypes.pointer(dub)
ChapelLib.func(doubPtr)
```

ChapelLib.chpl

```
export
proc func(x:c_ptr(real)) {…}
```

© 2019 Cray Inc.

Chapel Python Modules: This Effort

- Made Python module compilation use PrgEnv wrappers on Crays
  - It is better to use the Cray wrappers rather than the unwrapped compilers
    - E.g. 'PrgEnv-gnu' and 'cc' rather than just 'gcc'
  - Allowed builds with CHPL_REGEXP=re2, for instance
- Hid a warning with recent versions of Cython
  - Due to Python 2 deprecation — we require Python 3 anyways

© 2019 Cray Inc.

Chapel Python Modules: Next Steps

- Extend support to include calls to multilocale Chapel libraries
- Improve support for default argument values
- Performance improvements
  - Remove unnecessary copies for arrays and strings
- Support Anaconda distribution
  - Common among scientists/engineers/HPC users
- Error message improvements

© 2019 Cray Inc.

Chapel Python Modules: Next Steps

- Fix known bugs
  - Shutting down the Chapel runtime also ends Python execution
  - Python output lost when redirecting program output into a file

- Automatically set up and tear down runtime w/o user calls(?)
  - Remove need for 'chpl_setup()' and 'chpl_cleanup()' calls

- Remove fix for Python 2 deprecation warning when no longer needed

© 2019 Cray Inc.

LLVM for Libraries

© 2019 Cray Inc.

LLVM: Background and This Effort

Background: '--library' compilation did not work with the LLVM compiler backend

- Would still create an executable instead of a library

This Effort: fix basic compilation with '--library'

- Some issues remain:
  - Not all '--library-*' flags supported yet
  - Arrays encounter memory issues due to known ABI incompatibility

- Note: compiling C code w/ library built using LLVM requires listing it twice

```
… -Llib/ -lmyChplLib <chpl support libraries> -lmyChplLib
```

© 2019 Cray Inc.

LLVM: Next Steps

- Fix known issues:
  - Array memory issues
  - '--library-makefile'
  - '--library-python'
  - 'CHPL_LIB_PIC' not respected

© 2019 Cray Inc.

Position-Independent Code for Libraries

© 2019 Cray Inc.

PIC Library Code: Background

- Python requires linking against position independent code
  - So added 'CHPL_LIBMODE' for previous Python interoperability solution
    - Enabled use of '-fPIC' when building Chapel
    - Settings were "shared" or "none"
  - Runtime/third-party libraries wouldn't automatically rebuild for new settings
    - Had to fully rebuild whenever the setting was switched

© 2019 Cray Inc.

PIC Library Code: This Effort

- Replace with 'CHPL_LIB_PIC'
  - Settings are "none" or "pic”
    - "none" is the default
    - “pic” may impact performance, use sparingly
  - Runtime/third-party rebuilds when new setting is used
    - Old builds are kept around so don't have to rebuild every time

© 2019 Cray Inc.

Multilocale Libraries

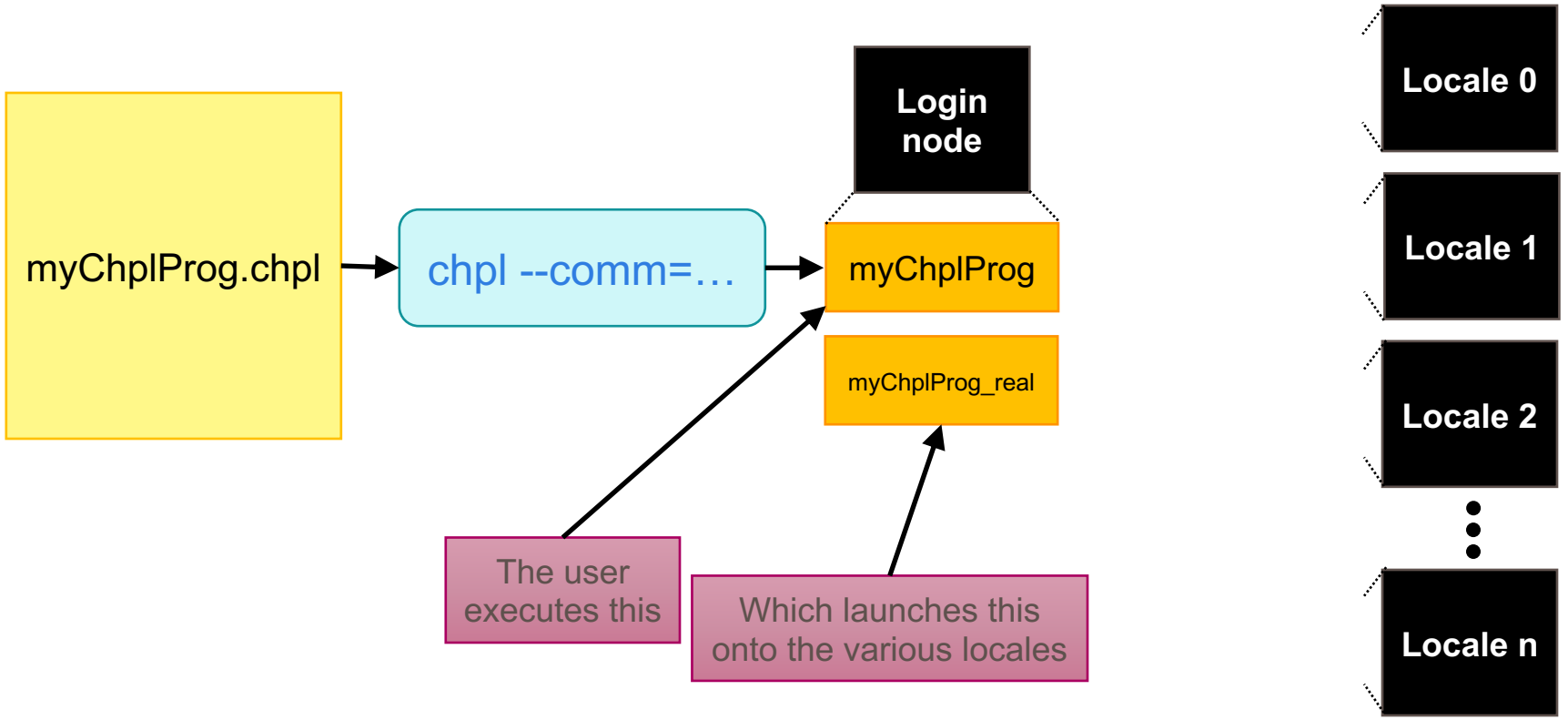
© 2019 Cray Inc.

Multilocale Libraries: Background

- Not automatically supported today
  - Chapel launcher does not adjust for wrapping a library file
  - Chapel expects to control how the program is distributed
    - Trickier when Chapel doesn't own 'main()'
  - Have mock-up of intended strategy
    - Users can and have implemented it themselves, but it's a lot of work
    - We're hoping to start adding automatic support in the next release cycle

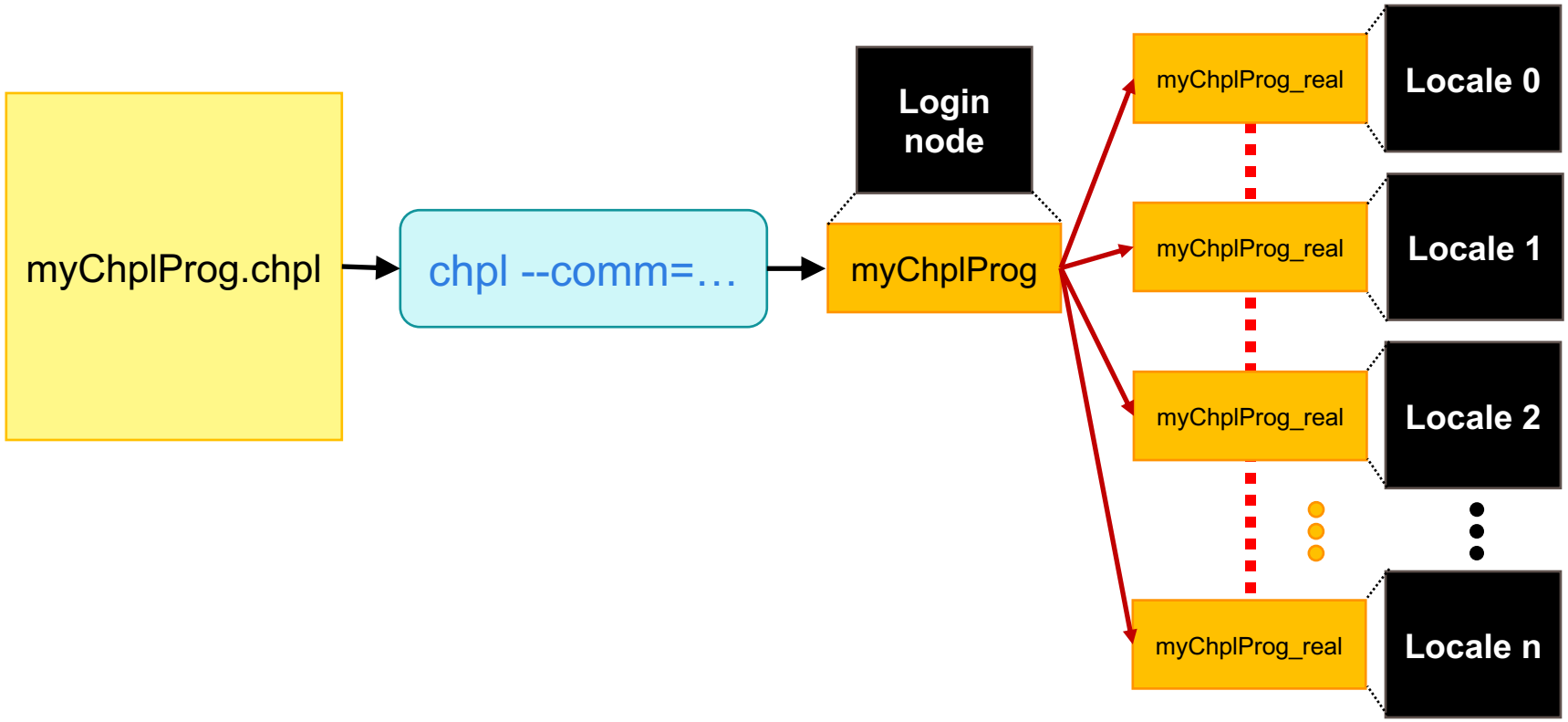
© 2019 Cray Inc.

Typical Multi-Locale Compilation + Execution



© 2019 Cray Inc.

Typical Multi-Locale Compilation + Execution



© 2019 Cray Inc.

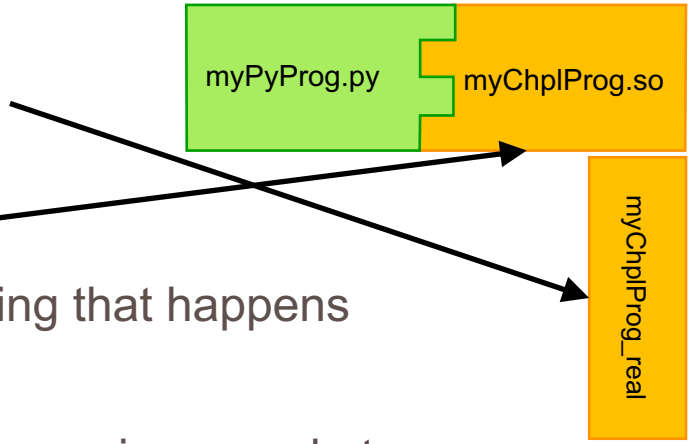
Multilocale Libraries: Background Motivation

- Make multilocale '--library' support feel natural
  - Like non-library multilocale, users shouldn't have to worry about launching
    - Should be agnostic to user program running on compute or login node
  - Similar behavior for function calls, argument types, etc. to single-locale
    - Though willing to change library initialization, e.g. for a numlocales arg

```
chpl_library_init(…); // Call to set up library in single-locale
chpl_multilocale_library_init(…, numLocs); // potential new call
```

© 2019 Cray Inc.

Multilocale Libraries: The Plan



- Will generate stand-alone binary to be launched
  - Its existence should be a black box to the user
    - Will be used by the library's interface
    - But the user won't be responsible for ensuring that happens
  - Library's interface will communicate to this binary using a socket
    - Binary's 'main()' will wait for function calls from socket, then execute them
    - Plan to use ZeroMQ module for communication
      - Some extensions needed, currently in progress

© 2019 Cray Inc.

Multilocale Libraries: The Plan

- User code will link to a library that launches the binary on initialization
  - Library will have wrappers that communicate to the launched binary

```
export proc foo(arg: int): int {  // definition in library file
    socket.send(/* actual foo's function number */);
    socket.send(arg);
    return socket.receive(int);
}
```

© 2019 Cray Inc.

Multilocale Libraries: The Plan

- Potentially implement a protobuf module for serialization
  - Instead of communicating arguments individually, can serialize into one
  - Could allow users to sidestep temporary argument type restrictions
    - E.g. to send class instance (which can't be an exported function arg today)
    - Users would serialize the instance themselves, then send and unpack
  - Protobuf is a well-known and widely used package
    - So having an implementation for Chapel is beneficial on its own merits

© 2019 Cray Inc.

Multilocale Libraries: Next Steps

- Short-term: Implement basic plan
  - Split user source file into library and executable sub-components
  - Implement communication between both sides
  - Only support string arguments for first draft

- Medium-term: Start work on serialization strategy
  - Extend argument types to all types single-locale interoperability supports

© 2019 Cray Inc.

Next Steps for Libraries

© 2019 Cray Inc.

Chapel Libraries: Next Steps

- Support multi-locale library builds natively
- Improve support for arrays
  - Multidimensional rectangular arrays
- Improve support for LLVM back-end
  - Expected to be the default soon
- Expand set of exportable features (e.g., additional types, variables, functions?)

© 2019 Cray Inc.

Interoperating with C Arrays

© 2019 Cray Inc.

c_array: Background

- Interoperability support had trouble with fixed-sized C arrays
- Compiler considered them as 'c_ptr', but this view is not quite accurate
  - A fixed-size C array field is allocated inside the structure
  - But a 'c_ptr' field is a pointer to some memory
- This approximation includes several drawbacks:
  - Missing opportunity for writeln of the fixed-size array to print elements
  - Setting a fixed-sized array field to a 'c_ptr' is invalid C code
    - Can occur within compiler-generated copy and default initializers

© 2019 Cray Inc.

c_array: This Effort

- 'c_array' is a new type in CPtr.chpl
- 'c_array'
  - Is a value type
  - Is always 1-dimensional
  - Stores the size as a 'param' compile-time constant
  - Indexing starts at 0
- Chapel compiler translates to a fixed-size array:

```
var x: c_array(c_int, 4)
// produces this C declaration
int x[4];
```

© 2019 Cray Inc.

c_array: Basic Usage

```
var arr: c_array(int, 4);
writeln(arr); // [0, 0, 0, 0]
for i in 0..3 do
  arr[i] = i;
writeln(arr); // [0, 1, 2, 3]
var copy = arr; // copy-initialize the elements
writeln(copy); // [0, 1, 2, 3]
copy[0] = 100; // modifies 'copy' but not 'arr'
writeln(copy); // [100, 1, 2, 3]
writeln(arr); // [0, 1, 2, 3]
```

© 2019 Cray Inc.

c_array: Interoperability

```
// header.h
typedef struct {
  double dat[2];
} mycomplex;

// test.chpl
extern record mycomplex {
  var dat: c_array(real(64), 2);
}
```

© 2019 Cray Inc.

c_array: Impact, Next Steps

Impact:

- Certain C interoperability scenarios are enabled
- Chapel includes a new useful low-level array type

Next Steps:

- Adjust compiler to use 'c_array' to implement homogeneous tuples
  - e.g. ‘4096*int’

© 2019 Cray Inc.

Renaming Extern Variables

© 2019 Cray Inc.

Rename Extern Vars: Background & This Effort

Background:

- Chapel's 'extern' declarations refer to external code (e.g., from C)
- Strings can be used to indicate that the external name differs from Chapel's

```
// This routine is called 'foo' in C, but 'c_foo' in Chapel
extern "foo" proc c_foo(x: c_int);
```

- This renaming has been supported for procedures and records

This Effort:

- Extend the renaming support to variables (including constants and fields)

```
extern "type" var c_type: c_int;
```

© 2019 Cray Inc.

Rename Extern Vars: Impact and Next Steps

Impact:

- Extends flexibility of renaming external symbols for use within Chapel
- Supports referring to external variables that conflict with Chapel keywords
- Helped with supporting Fortran array interoperability
  - Fortran's array descriptor has a field named 'type' (reserved in Chapel)

Next Steps:

- Extend renaming capability to support extern types
- Ensure all extern renamings work with general 'param' string expressions
  - Currently, only procedures do; others only support string literals

© 2019 Cray Inc.

For More Information

For a more complete list of related changes in the 1.19 release, refer to the 'Portability' and 'Interoperability' sections in the CHANGES.md file.

© 2019 Cray Inc.

SAFE HARBOR STATEMENT

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts.

These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

© 2019 Cray Inc.

THANK YOU

QUESTIONS?


chapel_info@cray.com


@ChapelLanguage


chapel-lang.org


cray.com


@cray_inc


linkedin.com/company/cray-inc-