Calling Chapel Code: Interoperability Improvements

Lydia Duncan and David Iten

CHIUW 2019

June 22, 2019

lydia@cray.com,
diten@cray.com

chapel-lang.org

@ChapelLanguage

Interoperability: Introduction

- Have had a draft capability to create libraries from Chapel source files
  - Compiling with '--library' flag generates library instead of executable
  - Historically designed for use from C
  - Given very little attention until recently

- Ideally, would allow integration of Chapel code into already large projects, e.g.
  - Enable easier distribution/parallelism for C or Fortran programs
  - Enable better performance for Python
    - Written in friendlier language than C, making it easier to debug

Interoperability: Introduction

- Accessible symbols specified via 'export' keyword

```
// Declares a Chapel function for use from outside the library
export proc foo(): int {…}
```

  - Only supports functions with concrete signatures
    - Exporting module-level variables or type definitions is future work

Interoperability: MonteCarloPi

- For this talk, we'll define three MonteCarloPi implementations via exported procs
  - We will demonstrate calling these functions from Python, C, and Fortran

MonteCarloPi.chpl:

```
// Computes pi using a serial implementation of the Monte Carlo simulation
export proc serialVersion(n: int, seed: int) {…}
// Computes pi using a task parallel implementation
export proc taskParVersion(n: int, seed: int) {…}
// Computes pi using a data parallel implementation
export proc dataParVersion(n: int, seed: int) {…}
```

Python Interoperability

Interoperability: Python

- Some background: Python support was provided via Simon Lund's PyChapel
  - Chapel code usable via inline doc strings, source files, fn body files

```
from pych.extern import Chapel
@Chapel()
def serialVersion(n = int, seed = int):
    """ … // Chapel code in Python comment """
    return None
```

  - Installed via pip, or by downloading and building the repository
    - Rather brittle: assumed Linux, virtual environment, Python 2 …
  - Didn't support most Chapel settings

Interoperability: Python

- We learned a lot, but decided to try a different tactic built using Cython
  - User doesn't have to write wrapper code themselves
    - Just compile library with '--library-python' and ensure on '$PYTHONPATH'

```
import MonteCarloPi // Treated like any other Python module!
```

  - Can also take advantage of Python's argument default values

```
// Both calls behave the same!
MonteCarloPi.taskParVersion(100000)
MonteCarloPi.taskParVersion(n=100000, seed=589494289)
```

C Interoperability

Interoperability: C

- C interoperability was a huge pain
  - Had to write header files/prototypes by hand, after inspecting generated C

MonteCarloPi.chpl:

```
export proc serialVersion(n: int, seed: int) { … }
```

MonteCarloPi.h:

```
#include "stdchpl.h"

void chpl__init_MonteCarloPi(int64_t _ln,
                             int32_t _fn);
void serialVersion(int64_t n, int64_t seed);
```

Interoperability: C

- Getting the '-I' includes and '-L'/'-l' libraries right for compilation was tricky

```
lydia@C02SY01RGTFM:exportArray (master)$ clang -fno-strict-overflow -I$CHPL_HOME/third-party/qthread/install/darwi
n-clang-native-flat-jemalloc-hwloc/include -I$CHPL_HOME/third-party/hwloc/install/darwin-clang-native-flat/include
 -DCHPL_JEMALLOC_PREFIX=chpl_je_ -DCHPL_HAS_GMP -fPIC -I$CHPL_HOME/modules/standard -I$CHPL_HOME/modules/packages
-Wno-unused -Wno-uninitialized -Wno-pointer-sign -Wno-tautological-compare -I$CHPL_HOME/third-party/qthread/instal
l/darwin-clang-native-flat-jemalloc-hwloc/include -I. -I$CHPL_HOME/runtime//include/localeModels/flat -I$CHPL_HOME
/runtime//include/localeModels -I$CHPL_HOME/runtime//include/comm/none -I$CHPL_HOME/runtime//include/comm -I$CHPL_
HOME/runtime//include/tasks/qthreads -I$CHPL_HOME/runtime//include/threads/none -I$CHPL_HOME/runtime//include -I$C
HPL_HOME/runtime//include/qio -I$CHPL_HOME/runtime//include/atomics/intrinsics -I$CHPL_HOME/runtime//include/mem/j
emalloc -I$CHPL_HOME/third-party/utf8-decoder -I$CHPL_HOME/runtime//../build/runtime/darwin/clang/arch-native/loc-
flat/comm-none/tasks-qthreads/tmr-generic/unwind-none/mem-jemalloc/atomics-intrinsics/hwloc/re2/fs-none/include -I
$CHPL_HOME/third-party/jemalloc/install/darwin-clang-native/include -I$CHPL_HOME/third-party/gmp/install/darwin-cl
ang-native/include -I$CHPL_HOME/third-party/hwloc/install/darwin-clang-native-flat/include -o callFuncReturnsArray
 callFuncReturnsArray.test.c -Llib/ -lreturnExternArray-L$CHPL_HOME/third-party/qthread/install/darwin-clang-nativ
e-flat-jemalloc-hwloc/lib -Wl,-rpath,$CHPL_HOME/third-party/qthread/install/darwin-clang-native-flat-jemalloc-hwlo
c/lib -L$CHPL_HOME/third-party/jemalloc/install/darwin-clang-native/lib -Wl,-rpath,$CHPL_HOME/third-party/jemalloc
/install/darwin-clang-native/lib -L$CHPL_HOME/third-party/gmp/install/darwin-clang-native/lib -Wl,-rpath,$CHPL_HOM
E/third-party/gmp/install/darwin-clang-native/lib -L$CHPL_HOME/third-party/hwloc/install/darwin-clang-native-flat/
lib -Wl,-rpath,$CHPL_HOME/third-party/hwloc/install/darwin-clang-native-flat/lib -L$CHPL_HOME/third-party/re2/inst
all/darwin-clang-native/lib -Wl,-rpath,$CHPL_HOME/third-party/re2/install/darwin-clang-native/lib -L$CHPL_HOME/lib
/darwin/clang/arch-native/loc-flat/comm-none/tasks-qthreads/tmr-generic/unwind-none/mem-jemalloc/atomics-intrinsic
s/hwloc/re2/fs-none -lchpl -lm -lgmp -ljemalloc -lchpl -lqthread -L$CHPL_HOME/third-party/hwloc/install/darwin-cla
ng-native-flat/lib -lhwloc -lm -lre2 -lpthread
```

- Had a shortcut to help, but didn't account for program-specific 'require's

```
[redacted]:exportArray (master)$ `$CHPL_HOME/util/config/compileline --compile` -o callFuncReturnsArray ca
llFuncReturnsArray.test.c -Llib/ -lreturnExternArray `$CHPL_HOME/util/config/compileline --libraries`
```

Interoperability: C

- Most of that information could be determined by the compiler, so generate it

```
chpl --library MonteCarloPi.chpl  # generates MonteCarloPi.h
# also creates helper Makefile, which includes program-specific information
chpl --library-makefile MonteCarloPi.chpl
```

- Compiling with the helper Makefile's variables is much easier

Fortran Interoperability

Interoperability: Fortran

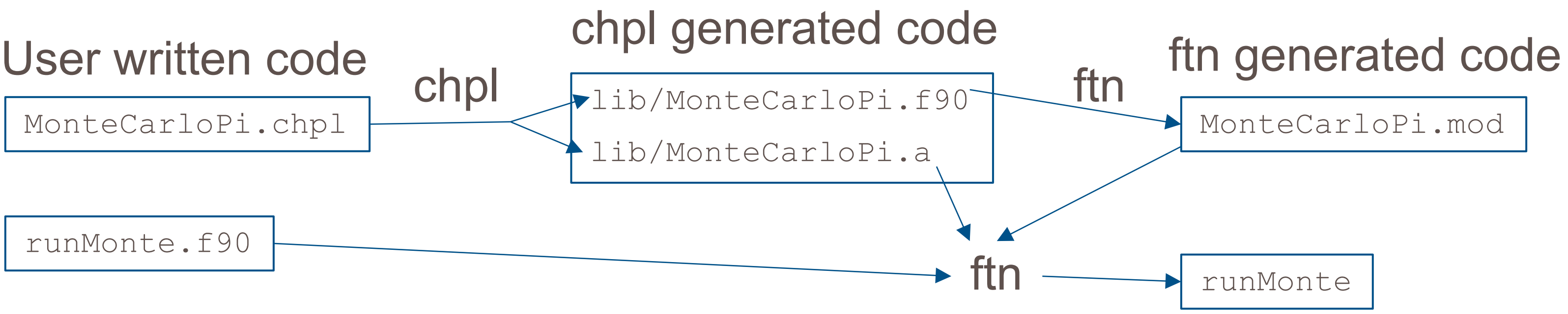
- Chapel compiler generates Fortran interfaces to Chapel libraries
  - Compiling with '--library-fortran' flag generates a library and interface module
  - Interface module is 'use'd by Fortran code to access Chapel symbols

```
program CallMonteCarloPi
  use MonteCarloPi                 ! use the generated library
  integer(8):: n, seed
  n = 100000
  seed = 589494289
  call chpl_library_init()         ! initialize the library
  call taskParVersion(n, seed)     ! call a Chapel procedure
  call chpl_library_finalize()     ! tear down the library
end program CallMonteCarloPi
```

Interoperability: Fortran

- Steps to compile and use the library

```
% chpl --library-fortran MonteCarloPi.chpl # build library and interface module
% gfortran -c lib/MonteCarloPi.f90          # create the module description file
% gfortran runMonte.f90 -Llib -lMonteCarloPi          \
    `$CHPL_HOME/util/config/compileline --libraries` \
    -o runMonte                             # compile and link the code and library
```



Interoperability Using Arrays

Interoperability: Arrays

- 1D arrays translate into native Chapel arrays from C, Fortran, and Python
  - Support parallel operations, slicing, etc.

```
export proc serialVersion(n: int, seed: int, p: [] real) { … p[i] = rnd; … }
```

```
import MonteCarloPi as mcp
A = [0.0]*200000
n = 100000; seed = 12345
…
mcp.serialVersion(n, seed, A)
```

Python

Fortran

```
use MonteCarloPi
real(8), dimension(200000) :: A
integer(8) :: n, seed
…
call serialVersion(n, seed, A)
```

```
#include "MonteCarloPi.h"
double* A = malloc(…);
int n, seed;
…
chpl_external_array Aext =
  chpl_make_external_array_ptr(A, n*2);
serialVersion(n, seed, Aext);
```

C

Interoperability: Arrays

- Chapel arrays without native representations handled opaquely in C or Python
  - e.g. Block or Cyclic Distributed arrays

```
export proc serialVersion(n: int, seed: int) {
  var A = newBlockArr(1..2*n);
  … A[i] = rnd; …
  return A;
}
```

Python

```
import MonteCarloPi as mcp
n = 100000; seed = 12345
A = mcp.serialVersion(n, seed)
… mcp.otherFunction(A) …
A.cleanup()
```

C

```
#include "MonteCarloPi.h"
int n = 100000, seed = 12345;
chpl_opaque_array A;
A = serialVersion(n, seed);
… otherFunction(A); …
cleanupOpaqueArray(&A);
```

What's Next?

Interoperability: Next Steps

CRAY

- Multilocale libraries
  - Launch multilocale Chapel library on compute notes
  - Communicate with it from the main program to execute functions
- Multidimensional arrays
  - Currently only support 1D arrays
  - Desire capability to support Chapel's rich multidimensional arrays
- Support calling Chapel libraries from additional languages
  - C++
  - Chapel
  - More?

Multilocale Libraries

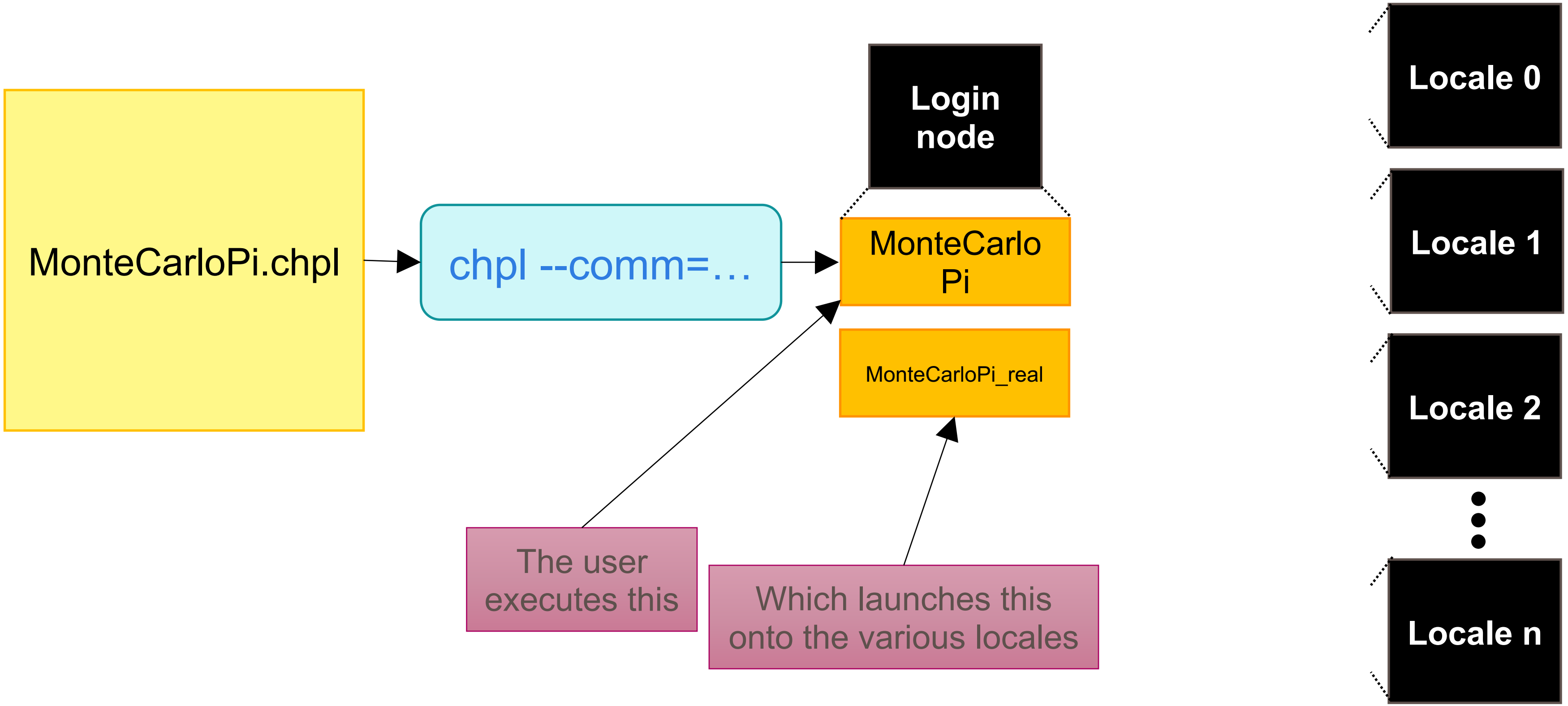
Multilocale Libraries: Background

- Not automatically supported
  - Chapel launcher did not adjust for wrapping a library file

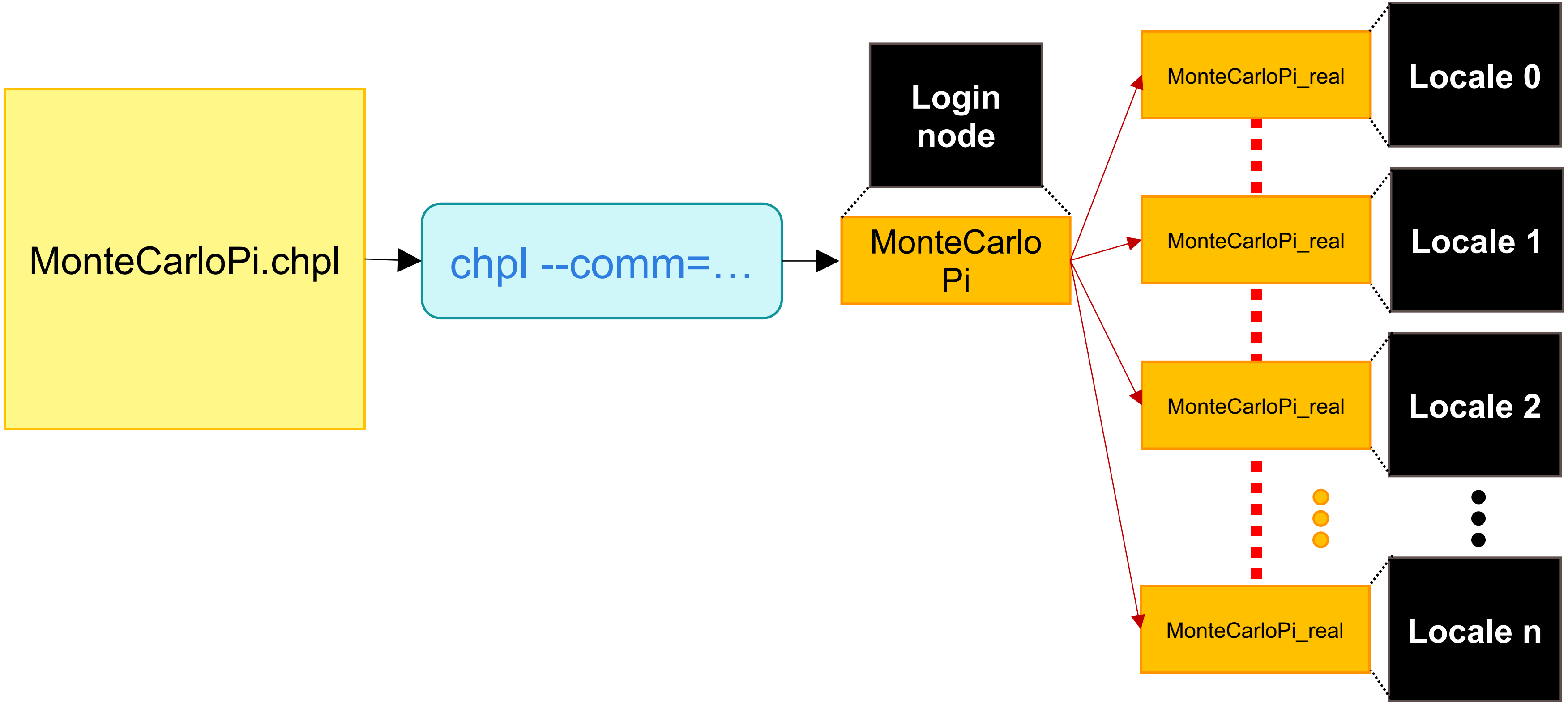
  - Chapel expected to control how the program is distributed
    - Trickier when Chapel doesn't own 'main()'

  - Had mock-up of intended strategy
    - Users can and have implemented it themselves, but it's a lot of work
    - We've started adding automatic support but there are still some kinks
      - Should have a basic version in 1.20

Typical Multi-Locale Compilation + Execution



Typical Multi-Locale Compilation + Execution



Multilocale Libraries: Background Motivation

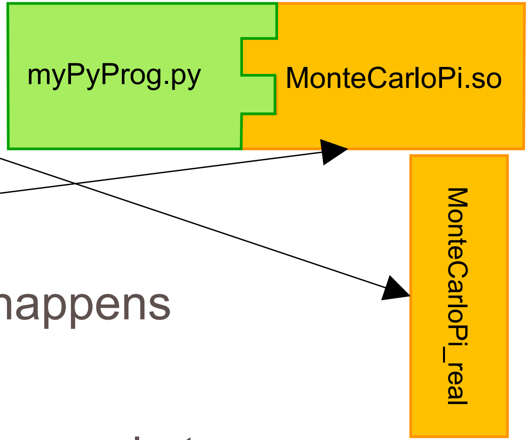
- Make multilocale '--library' support feel natural
  - Like non-library multilocale, users shouldn't have to worry about launching
    - Should be agnostic to user program running on compute or login node

  - Similar behavior for function calls, argument types, etc. to single-locale
    - Though we do expect a numlocales argument for library initialization

```
chpl_setup(); // Call in Python to set up library in single-locale
chpl_setup(numLocales); // New call
```

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
    - Use ZeroMQ module for communication
      - Some extensions needed, currently on master

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

```
// definition in library file
export proc serialVersion(n: int, seed: int) {
    socket.send(/* actual serialVersion's function number */);
    socket.send(n);
    socket.send(seed);  // we hope to bundle these args together eventually
    socket.receive(/* type of indicator that the function is done */);
}
```

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

Multilocale Libraries: Next Steps

- Short-term: Finish implementing basic plan
  - Have split user source file into library and executable sub-components
  - Have implemented communication between both sides
  - Supports c_string arguments and most other primitives
    - Still need to support arrays
    - We're also looking into supporting Chapel strings

- Medium-term: Start work on serialization strategy

And that's it!

Interoperability: Additional Resources

- We have a technote describing the currently supported features
- We intend to create a primer or two to demonstrate using the generated libraries
- And of course, you can ask questions on our mailing lists, Gitter channel, StackOverflow, etc.

Interoperability: Acknowledgements

- We would like to thank Ben Albrecht, Brad Chamberlain, Michael Ferguson, David Longnecker and the rest of the Chapel team, past and present, for their feedback, input and contributions.
- We would also like to thank:
  - Simon Lund for his PyChapel implementation
  - Russel Winder for his continued interest and encouragement of Python interoperability
  - And various other users that have also provided their feedback

- Cython and Fortran's ISO/IEC TS 29113 were indispensable for this effort.

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.

THANK YOU

QUESTIONS?

lydia@cray.com
diten@cray.com

@ChapelLanguage

chapel-lang.org