Ongoing Efforts

Chapel Team, Cray Inc.

Chapel version 1.18

September 20, 2018

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.

Outline

- Open Fabrics Interface ('ofi') Communication Layer
- Creating and Using Chapel Libraries

Open Fabrics Interface (‘ofi’) Communication Layer

‘ofi’ Comm Layer: Background and This Effort

Background: Progress toward an OFI-based comm layer

- Goal (dream?): a single comm layer supporting all HPC networks, and with timely performance
- Previously: “Design work and and stubbed implementation complete”
- Turned out to be premature
- Encountered problems expanding the stubbed implementation

This Effort: ofi “mock-up”

- Standalone multi-node proxy for comm layer activities
  - Registers memory, sends & handles Active Messages, does RDMA, etc.
  - Small: functional portion only 1/10 LOC of comm=ugni
- Avoided comm layer intricacies while prototyping network interactions
- Quicker exploration cycle (study ⇒ code ⇒ test)
- Completed in mid-September

‘ofi’ Comm Layer: Impact, Status, Next Steps

Impact: Path to comm=ofi is clear

- Have match between comm layer needs and provider capabilities
- Working code demonstrates basic comm layer functions (AM, RDMA)

Status: Functionality sufficient, performance adequate

- Mockup works with both sockets and gni providers
- Single-thread performance compared to comm=ugni:
  10% AM rate, 50% RDMA bandwidth; ok for now

Next Steps: Produce initial comm=ofi implementation

- Adapt code for network interactions (AM, RDMA) from ofi mockup
- Adapt code for runtime interactions (tasking, e.g.) from comm=ugni

Creating and Using Chapel Libraries

Chapel Libraries: Outline

- Background
- Chapel Code Changes
- Calling from C
- Python Modules
- Arrays
- Error Message Improvements
- Status and Next Steps

Chapel Libraries: Background

- **Have had a draft capability to create Chapel libraries**
  - Historically designed for use from C
  - Left much to be desired…

- **Accessible symbols specified via `export` keyword**

```
export proc bar(): int { … }
```

  - Only supports exporting functions with concrete signatures
    - Couldn't export functions involving array arguments (considered generic)
    - Can't export module-level variables or type definitions

Chapel Libraries: Chapel Code Changes

Chapel Changes: Background & This Effort

Background:

- Module-level variables were not initialized in library mode
  - Could be referenced by exported functions
  - But, would not have been given initial value

This Effort:

- Automatically export module initialization functions
  - For a module ‘foo’, creates routine named `chpl__init_foo()`
  - Establishes initial values of module-level variables
    - `chpl_library_finalize()` call deinitializes such variables

Chapel Changes: Next Steps

- **Allow multiple Chapel libraries to be used by one program**
  - Currently, each library includes the Chapel runtime
    - Linking multiple libraries leads to duplicate symbols
- **Create single entry-point to initialize modules and runtime**
  - Similar to Python support described in subsequent slides
  - Or even zero calls to set things up?
- **Support exporting module-level variables, types**

Chapel Libraries: Calling From C

Calling From C: Background

- **Client programs must call two runtime functions …**

  … one to set up the Chapel runtime and third-party libraries …

  ```
  void chpl_library_init(int argc, char* argv[]);
  ```

  - Must be called prior to any calls in the generated library itself

  ... and one to clean up at the end of the program

  ```
  void chpl_library_finalize(void);
  ```

Calling From C: Background

- **Generated library using `--library`**
  - For foo.chpl, `--dynamic` created `foo.so` and `--static` created `foo.a`
    - Default behavior determined by platform, back-end compiler
  - Could change name using `-o`/`--output` flag

```
chpl --library -o libfoo foo.chpl # libfoo.a or libfoo.so
```

Calling From C: Background

- **Header files / prototypes had to be written by hand**
  - Had to inspect generated C code for Chapel→C translation

myLib.chpl:

```
export proc foo(x: int): int { … }
```

myLib.h:

```
#include "stdchpl.h"

void chpl__init_myLib(int64_t _ln,
                      int32_t _fn);
int64_t foo(int64_t x);
```

Calling From C: Background

- **Compilation command to use libraries was very extensive**
  - Needed to include runtime and third-party directories

```
[redacted] 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
```

  - Even when using \`compileline\` shortcut, still longer than ideal
    - also, doesn't account for \`require\` statements in the code

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

Calling From C: This Effort

- **Improved the naming of the generated library**
  - Prepends “lib”, unless name already started with “lib”

```
chpl --library foo.chpl          # libfoo.a
chpl --library libfoo.chpl       # libfoo.a
chpl --library -o bar foo.chpl   # libbar.a
```

- **Started generating a header file alongside the library**
  - Default name comes from base library name
  - Can change using `--library-header`

```
chpl --library foo.chpl                             # generates foo.h
chpl --library -o bar foo.chpl                      # generates bar.h
chpl --library --library-header bar foo.chpl        # generates bar.h
chpl --library-header bar foo.chpl                  # generates bar.h
```

Calling From C: This Effort

- **Added `--library-makefile` to generate a Makefile stub**
  - Named `Makefile.<base library name>`
  - Defines Makefile variables for:
    - Compilation flags and include directories (`CHPL_CFLAGS`)
    - Library directories and `-l` libraries (`CHPL_LDFLAGS`)
    - The back-end C compiler used to create the library (`CHPL_COMPILER`)
    - Linker commands (`CHPL_LINKER` and `CHPL_LINKERSHARED`)
  - Can be included by other Makefiles to simplify compilation
    - Sample Makefile for `foo.chpl` and client C code `myCProg.c`:

```
include lib/Makefile.foo

myCProg: myCProg.c lib/libfoo.a
  $(CHPL_COMPILER) $(CHPL_CFLAGS) -o myCProg myCProg.c $(CHPL_LDFLAGS)
```

Calling From C: This Effort

- **Changed the default location of the generated files**
  - Was: same directory as compilation command
  - Now: defaults to “lib/” sub-directory (will create if it doesn’t exist)
  - Can change location via `--library-dir` flag

```
chpl --library --static foo.chpl # lib/libfoo.a, lib/foo.h
chpl --library --static --library-dir bar foo.chpl # bar/libfoo.a ..
```

- **All `--library-*` compilation flags implicitly throw `--library`**
  - `--library-header`
  - `--library-makefile`
  - `--library-dir`
  - And the Python library flags (see upcoming slides)

Calling From C: This Effort

- **Reflect Chapel `require` statements in C and Makefiles**
  - Headers result in a `#include` in generated .h files

    `require "bar.h"` → **#include** "bar.h"

  - Libraries get added to the generated Makefile's `CHPL_LDFLAGS`

    `require "-lbar"` → CHPL_LDFLAGS = … -lbar …

Calling From C: Impact

- **--library compilation is now easier to use**
  - Users have less repetitive code to write
  - Generated Makefile makes compiling with generated libraries easier

- **Library name is now more standard**

- **Functionality is expanded**
  - Module-level variables now have their declared initial values

Chapel Libraries: Python Modules

Python Modules: Background

- **Python interoperability was provided through PyChapel**
  - The implementation was prototypical
    - Contributed from the open-source community
  - Supported some primitive types and 1D arrays of reals
    - Multidimensional arrays and arrays of other types not supported

  - Chapel code usable via inline doc strings, source files, fn body files
    - Inline example:

```
from pych.extern import Chapel
@Chapel()
def hello_world():
  """
  writeln("Hello, world");
  """
  return None
```

Python Modules: Background

- **PyChapel was hard to use and hard to maintain**
  - Installed via pip, or by downloading and building the repository
    - Installation process rather brittle: assumed Linux, virtual environment …
    - Also assumed a particular directory structure
  - Only worked for Python 2, not Python 3
  - Required quickstart settings for Chapel
    - No qthreads, no jemalloc …

Python Modules: This Effort

- **Added support for a new compiler flag `--library-python`**
  - Generates and compiles Cython files under the hood
- **Accessible via normal Python `import` and function calls**
  - Directory with generated files must be in `$PYTHONPATH`
- **Supports all Chapel primitives, C strings, 1D arrays**
  - Primitives of different sizes (e.g. `int(8)`) supported via NumPy
  - C strings correspond to Python `bytes` type
  - 1D array arguments supported via anything iterable
  - 1D array returns supported using NumPy arrays

Python Modules: This Effort

- **Supports Python 3**
  - Decided not to support Python 2 for now
    - Python 2 support expected to end after 2020
- **Works for any single-locale Chapel installation**
  - Multi-locale support designed and prototyped, but not implemented
- **Name of generated module matches base name of library**
  - foo.chpl can be used via `import foo` by default
  - Can change module name (without changing the .a/.so name):
    - `--library-python-name`
    - Turns on creation of the Python module if not already specified

```
chpl --library-python-name foo foobar.chpl # Python module: foo
```

Python Modules: This Effort

- **As in C, user must set up and tear down Chapel runtime**
  - Unlike C, no need for a separate call to module initialization function

```
import foo              // Import Chapel module

foo.chpl_setup()        // Set up Chapel runtime, third party libs, module-level vars
foo.baz(7)              // Call into a library function
foo.chpl_cleanup()      // Shut down the Chapel runtime and exit the program
```

Python Modules: Status

- **PyChapel is now deprecated**
- **--library-python has more functionality than PyChapel**
  - Lives in Chapel repo rather than a distinct one
- **Plenty of work remains**
  - Yet, desired features seem achievable

Python Modules: Next Steps

- **Improve support for arrays and C strings**
  - Currently performs copies
  - Would like to access arrays in-place

- **Explore supporting default values for arguments**
  - C doesn’t support this
  - But the Python code that calls it could …

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
- **Support Anaconda distribution**
  - Common among scientists/engineers/HPC users
- **Error message improvements**

Chapel Libraries: Arrays

Arrays: Background

- **Couldn't export functions involving arrays**
  - Array arguments were considered generic, even when fully specified

    ```
    proc foo(x: [0..5] int) { … }
    ```

    - In Chapel, this routine accepts a 1D array with any domain map
    - But, generic routines can't be exported…

- **PyChapel supported 1D arrays of 'real' arguments**
  - Didn't support:
    - Returning arrays
    - Multidimensional arrays
    - Arrays of integers, bools, strings, …

Arrays: This Effort

- **Exported functions can take 1D dense array arguments**
  - Declared like normal Chapel functions

    ```
    export proc foo(x: [0..3] int): [0..3] int { … }
    ```

  - Domain must start at 0
  - Can omit domain declaration
    - C version of array will store size (see later slides on calling from C)
  - Cannot omit element type
    - No way to store without hard-coding it via C type
    - Argument would be generic (and can't export generic functions)

Arrays: This Effort

- **Exported functions can return 1D dense arrays**
  - Cannot omit return type declaration when returning arrays
    - Return type will not be properly transformed
  - Can omit the domain and/or element type, e.g.

```
export proc foo(…): [] { … }
```

    - Chapel will error when client code is run if inferred domain is inappropriate
    - Element type won't be visible in C, client will have to reason about it

Arrays: Calling from Python

- **Python users can call functions that take or return arrays**
  - Array arguments will accept any iterable Python object
    - Will copy contents at present
    - Have ideas about how to avoid this penalty
  - Returned arrays will be NumPy arrays

```
import intArrays

intArrays.chpl_setup()            # set up runtime, modules
x = [5, 4, 3, 2, 1]               # list of int
intArrays.takesArray(x)
y = intArrays.returnsArray()      # array of numpy.int64
intArrays.takesArray(y)
intArrays.chpl_cleanup()          # shut down Chapel code
```

Arrays: Calling Functions

- **Calling from the C side:**
  - Requires use of a wrapper struct for appropriate translations:

```
typedef struct {
  void* elts; // pointer to C array
  uint64_t size;

  chpl_free_func freer; // function to free the array memory, if applicable
} chpl_external_array;
```

  - chpl_external_array will assume the correct element type is used
    - Like any C program, memory errors will occur if this is not true

Arrays: Calling from C

- **Two ways to create instances of chpl_external_array**
  - From a pointer and the size of the buffer it points to:

```
chpl_external_array chpl_make_external_array_ptr(void* elts,
                                                 uint64_t size);
```

  - From the size and number of elements:

```
chpl_external_array chpl_make_external_array(uint64_t elt_size,
                                             uint64_t num_elts);
```

- **Its free function can be called via this helper:**

```
void chpl_free_external_array(chpl_external_array x);
```

  - Workaround for issue with C function pointers in Chapel code

Arrays: Impact

- **Storing the free function allows it to be called anywhere**
  - Using different allocation/free strategy can cause problems

```
void* alloc1 = chpl_mem_alloc(…);
free(alloc1); // doesn't tell Chapel the memory is free, could cause problems
void* alloc2 = malloc(…);
chpl_mem_free(alloc2 …); // tells Chapel to free memory it wasn't tracking!
```

  - If stored, user doesn't have to reason about which one was used
  - `x.freer == NULL` means someplace else will clean it up

Arrays: Impact

- **Wrapper replacement keeps direct 1:1 translation for args**
  - Chapel array argument doesn't turn into array + size
  - Chapel array return can communicate size with returned memory

- **This is a tradeoff between elegance in C vs. Chapel**
  - C must use chpl_external_array structure around native arrays
  - This design decision is still under active discussion in this issue

Arrays: Next Steps

- **Eliminate unnecessary array copies to compute in-place**

- **Add support for arrays that are:**
  - Multidimensional
  - Sparse
  - Distributed
  - Associative

- **Revisit design of chpl_external_array structure**
  - And its counterpart in Chapel module code

Chapel Libraries:

Error Message Improvements

Error Messages: Strings

Background:

- Functions involving strings were causing link-time issues

```
proc foo(x: string): string { … }
```

  - 'string' type defined entirely as Chapel code and not currently exportable
  - Wouldn't cause problems until library was linked
    - Could bite user without access to original code

- Can translate a C string into a Chapel string in Chapel code
  - Performing same operation at the C level has large potential for errors

This Effort:

- Temporary fix: generate compile-time error when using strings
  - Signals to library author to switch to `c_string` arguments / returns

Error Messages: Multiple Modules

Background:

- Generated error asking for '--main-module' flag when multiple modules
  - e.g., when two source files are included on the command line
- But main() has no meaning in library compilation
  - It causes a warning when included

This Effort:

- Only require '-o' / '--output' flag for libraries with multiple modules
  - Used to determine generated name (which would be difficult to determine)

```
chpl --library -o foo A.chpl B.chpl
```

Chapel Libraries: Status and Next Steps

Chapel Libraries: Status & Next Steps

Status:

- The library technote has been updated to reflect the new features
- Expanding current support remains a priority

Next Steps:

- Expand set of features
- Improve handling of arrays and strings *in situ*
- Add support for other languages:
  - Fortran
  - Chapel code using precompiled Chapel libraries
  - C++

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY