Library Improvements

Chapel Team, Cray Inc.

Chapel version 1.15

April 6, 2017

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

- **New Modules**
  - Date and Time
  - Owned and Shared
  - Futures
  - LinearAlgebra
- **Module Improvements**
  - BLAS Improvements
  - FFTW Improvements
  - MPI Improvements
  - Other Library Improvements

New Modules

Date and Time

Date and Time Module: Background

- **Desirable to work with dates and times from Chapel**
  - Including generating, manipulating and comparing them
- **No such functionality previously existed in Chapel**

Creative Commons Flickr/twak

Date and Time Module: This Effort

- **Implement a Date/Time module to handle the details**
- **Largely inspired by the Python datetime module**
- **Types to represent…**
  - …Times (`record time`)
  - …Dates (`record date`)
  - …Combined Dates and Times (`record datetime`)
  - …Amounts of time (`record timedelta`)
  - Abstract base class for time zones (`class TZInfo`)
- **Operators to combine and compare in useful ways e.g.**

```
datetime + timedelta ⇒ datetime
date - date ⇒ timedelta
timedelta / int ⇒ timedelta
datetime >= datetime ⇒ bool
```

Date and Time Module: Other Useful Methods

- **Constructor/Factory Methods**

```
[date|datetime].today()                    // the current date
[date|datetime].fromtimestamp(timestamp)   // the date for 'timestamp'
[date|datetime].fromordinal(ord)           // 'ord' days after 12-31-0000
datetime.now()                             // the current date and time
datetime.combine(date, time)               // combine the date and time
```

- **Formatting Methods**

```
[time|date|datetime].isoformat()           // create string
[time|date|datetime].strftime(formatStr)   // create string
datetime.strptime(dateStr, formatStr)      // read from string
```

- **General Methods**

```
[date|datetime].toordinal()        // number of days since 12-31-0000
[time|date|datetime].replace()     // Create a new value with fields replaced
[date|datetime].weekday()          // Day of the week for date
[date|datetime].isocalendar()      // (ISO year, ISO week #, ISO day of week)
```

Date and Time Module: Status and Next Steps

Status:

- Available in new DateTime standard module
- Allows users to store dates and times
- Manipulate, compare, and query information about them
- Includes basic support for including time zones
  - Time zone definitions not included
  - Can write 'TZInfo' sublcasses to implement time zones as needed

Next Steps:

- Further review of interface and naming taking user input into account

Owned and Shared

Owned and Shared: Background

- **Chapel doesn't have garbage collection (GC)**
  - Users have to explicitly 'delete' class instances
  - Traditional GC is unlikely to be appropriate for Chapel
- **How does GC compare?**

Garbage Collection	'delete'
+ simpler programming + eliminates memory leaks + eliminates common error cases	– more chances for programmer error – failure to delete results in leaks – may double delete, use-after-free
– implementation challenges due to distributed memory & parallelism	+ simpler implementation
– performance challenges – stop-the-world interrupts program – concurrent collectors add overhead – scalability may prove difficult	+ predictable, scalable performance

Owned and Shared: Background

- **Rust and C++ auto-pointers use a different strategy**
  - user manages *ownership*; implementation takes care of deleting
  - Rust includes compile-time checking to ensure safety properties
    - in particular, compiler proves no use-after-free

- **A related approach seems better for Chapel**
  - better usability than requiring 'delete'
  - better performance than traditional GC

- **Some Chapel types already use a similar approach**
  - involves *wrapper records*…

Owned and Shared: Wrapper Records

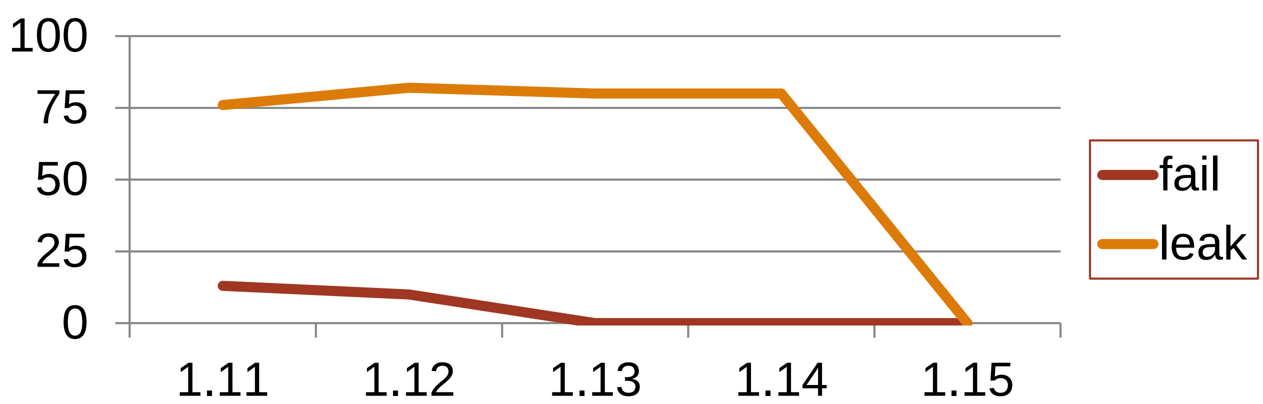
- **Wrapper records enable class memory management**
  - a class implements a particular data type
  - a record stores an instance of the class
  - the record controls copy and assignment behavior
    - copies can point to the same class instance, or
    - copies can allocate separate class instances, or …
  - the record's deinit() method handles deleting the class instance

- **This pattern is used with many built-in types**
  - e.g., domains, arrays, distributions, strings

- **Wrapper records rely on the implementation of records**
  - correct record initialization, copy, and destruction are key

Owned and Shared: Record Progress

- **Historically, records had memory management issues**
- **Fixing those has enabled progress in related areas:**
  - addressing leaks in record-wrapped types
  - implementing more types as records
- **Recent progress in design and implementation:**
  - design: CHIP 13 "When Do Record and Array Copies Occur"
  - implementation: graph below shows improvement in record tests



Owned and Shared: This Effort

- **Create general-purpose wrapper records**
  - building upon progress with records

- **Two initial patterns:**
  - **Owned:** uses a single-owner pattern to manage lifetime
    - deletes contained class instance when it goes out of scope
    - assignment and copy initialization are destructive ownership transfers
  - **Shared:** uses reference-counting to manage lifetime
    - contained class instance deleted when all Shared copies destroyed
    - assignment and copy initialization share ownership

- **Interested in feedback on this initial effort**

Owned and Shared: Usage

- **Create Owned or Shared types with a class instance:**

```
var myOwned  = new Owned(new MyClass());
var myShared = new Shared(new MyClass());
```

- **Empty an Owned or Shared and delete if appropriate**

```
myOwned.clear();  // may delete instance; leaves the record storing nil
```

- **Set the instance managed by an Owned or Shared**

```
myShared.retain(new MyClass());  // may delete previous instance
```

- ***Borrow* a pointer to the instance**

```
var instance:MyClass = myOwned.borrow();
// instance (the result of the borrow) is only valid while:
// 1) the Owned/Shared record contains that instance
// 2) the Owned/Shared record is in scope
```

- **Call a method on the class**

```
myShared.myClassMethod();  // forwards to borrow().myClassMethod()
```

Owned and Shared: Usage of Shared

- Starting with a Shared record managing a class instance:

```
var myShared = new Shared(new MyClass());
```

- Share ownership with assignment or copy-initialization:

```
var otherShared = myShared;
// now otherShared and myShared point to the same instance
// the instance will be deleted when all copies of the Shared go out of scope
// both assignment and copy-initialization share ownership
```

Owned and Shared: Usage of Owned

- **Starting with Owned records managing class instances:**

```
var myOwned      = new Owned(new MyClass());
var anotherOwned = new Owned(new MyClass());
```

- **Destructively transfer ownership:**

```
var otherOwned = anotherOwned;
// anotherOwned now stores nil
// both assignment and copy-initialization transfer ownership
```

- **Stop managing an instance and return it:**

```
var instance = myOwned.release();
// myOwned now stores nil and is no longer responsible for deleting;
// calling code must arrange to delete instance to prevent a memory leak
delete instance;
```

Owned and Shared: Safety Properties

- **Are memory leaks still possible?**
  - yes, un-managed class instances can be created and used
  - also, a class instance can be managed for only part of its lifetime
    - un-managed before it is provided to Owned / Shared
    - un-managed after Owned.release()

- **Is use-after-free possible?**
  - yes, but in the future it might be detected at compile-time
  - one use-after-free is possible in this way:
    - result of 'borrow' is stored in a global variable
    - Owned / Shared record goes out of scope and deletes the instance
    - the borrowed pointer is dereferenced
  - another possible use-after-free:
    - a class instance is created and stored in a global variable
    - Owned record initialized with it and is destroyed, deleting the instance
    - the global variable is dereferenced

Owned and Shared: nil Safety

- **Can an Owned / Shared record store nil?**
  - currently, yes
    - like a variable of class type
- **What happens with nil dereferences of class variables?**
  - philosophy: only erroneous programs can have nil dereferences
  - run-time checks for nil dereferences are available
  - these are disabled with --fast, --no-checks, or --no-nil-checks
- **Should Owned / Shared include more checking?**
  - e.g. compiler proves that nil Owned / Shared is never dereferenced
  - e.g. always-on checks for nil in 'borrow', 'retain', or 'release'
  - current answer: no
    - it would be a big break from existing class behavior and philosophy
    - preventing nil class instances has big impact on the language design
    - … e.g. must array elements be explicitly initialized on array creation?

Owned and Shared: Convenience

- **Can Owned(T) or Shared(T) pass to an arg:T formal?**
  - currently, no
  - we are considering allowing it with user-defined coercions

- **Can Owned(Child) coerce into Owned(Parent)**

  … assuming 'class Child : Parent' ?
  - currently, no
  - we are considering allowing it with user-defined coercions

- **Can a method on T be called directly on an Owned(T) ?**
  - currently, yes
  - uses the new 'forwarding' feature

Owned and Shared: Current Surprises

- **Forwarding, but not coercing generally, can be surprising:**

```
var a = new Owned(new C());
var b = new Owned(new C());
a.matches(b);

class C {
  proc matches(other) {
    return this == other; // error - this: C but other: Owned(C)
  }
}
```

  - could be addressed with support for coercion from Owned(T) to T

- **L-value checking is surprising for Owned:**

```
var myOwned: Owned(C);
myOwned = new Owned(new C(1)); // error: illegal lvalue in assignment
```

  - happens because Owned assignment is destructive (modifies RHS)
  - could be addressed by relaxing l-value rules for Owned or generally

Owned and Shared: Impact, Status, Next Steps

Impact:

- Easier to manage memory for class instances

Status:

- Owned, Shared are in package modules OwnedObject, SharedObject
- Interface is not yet final

Next Steps:

- Gain experience using Owned and Shared
- Address surprising l-value errors
- Decide if we want to implement compile-time use-after-free checks
  - may require significant language changes
  - see Borrow Checker in Rust and DIP 1000 in D
- Decide if we want to support coercions
  - from Owned(T) to T
  - from Owned(Child) to Owned(Parent)
  - if so, start by implementing user-defined coercions

Futures

Futures: Background

- **Futures are a frequently requested feature**
  - Futures for Chapel have been explored as far back as 2013
- **A Future…**

  …computes a function call in the background
  …is linked to a task to compute a value
  …can be stored in a variable
  …can be waited upon to return the value
- **Advantages over Chapel tasks and 'sync' / 'single' vars:**
  - programs using only immutable future variables are deadlock-free
  - runtime can know which task will unblock another
  - simpler way to write the pattern of tasks that produce a value

Futures: This Effort and Next Steps

This Effort: Added Futures package module

- Contributed by Nick Park

```
use Futures;
proc calculate(n) { … }
const future = async(calculate, 10); // starts calculate(10) in a task
// do other useful work…
compute(future.get()); // waits for task, passes result to compute()
```

Next Steps: Consider incorporating Futures into the language

- e.g., 'begin' expressions could generate Futures
- consider deprecating 'single' in favor of Futures

LinearAlgebra

LinearAlgebra: Background

- **Linear algebra is core to a large number of applications**
  - Machine learning, quantum chemistry, computational physics, etc.

- **Chapel's linear algebra support in 1.14 included:**
  - LAPACK module
    - Chapel interface to standard LAPACK library
  - BLAS module
    - Chapel interface to standard BLAS library
  - LinearAlgebraJAMA
    - Written natively in Chapel
    - Limited routine coverage

LinearAlgebra: This Effort

- **Design and implement a Chapel linear algebra library**

- **Current design choices**
  - Implement in terms of BLAS for performant computations
    - Will also utilize LAPACK in future versions
  - Use Chapel arrays as matrices and vectors
    - Allows interoperability between LinearAlgebra matrices and other modules
  - Matrix and vector initializers create arrays with 0-based domains
  - Make additional array methods available through the module
    - For example:

```
proc _array.T { return transpose(this); }
```

LinearAlgebra: Features

- **Matrix / Vector convenience initializers**

```
var v = Vector(4);      // vector
var m = Matrix(3, 4);   // matrix
var i = eye(10, 10)     // identity matrix
```

- **Matrix structure functions**

```
isDiag(A: [])
isHermitian(A: [])
isSymmetric(A: [])
...
```

- **Matrix/vector operations**

```
dot(A, B)  // for combinations of scalars/vectors/matrices
```

LinearAlgebra: Impact

Example 1: Rotate a vector with respect to Z-axis:

```
use LinearAlgebra;

var   v1     = Vector(1, 0, 0, eltType=real);
const theta = pi;

var Rz = Matrix([cos(theta), -sin(theta), 0.0],
                [sin(theta),  cos(theta), 0.0],
                [0.0,         0.0,        1.0],
                eltType=real);

var v2 = dot(Rz, v1);
```

LinearAlgebra: Impact

Example 2: Demonstrates initializers, dot, and transpose

```
use LinearAlgebra;
use Random;

var rs = new RandomStream(real);

var M1 = Matrix(1000, 1000),
    M2 = eye(1000, 1000);
rs.fillRandom(M1);

// M1.T == transpose(M1)
var M3 = dot(M1.T, M2);
```

LinearAlgebra: Status & Next Steps

Status: LinearAlgebra prototype available in Chapel 1.15

- Prototype-related caveats noted in documentation

Next Steps: Improve LinearAlgebra module

- More features
  - Aiming for feature-coverage similar to Matlab and NumPy
- Support LAPACK routines
- Further review of design tradeoffs, taking user input into account
- Sparse array support
- Distributed array support
- More efficient native algorithms
  - e.g. transpose

Module Improvements

BLAS Improvements

BLAS Improvements: Background

- **The BLAS module is made up of two components:**
  - **C_BLAS:** Low-level extern API
    - Submodule in BLAS
    - C-type arguments

```
extern proc cblas_dgemm(…TransA: c_int, M: c_int, N: c_int, …
                        A:[] c_double, …)
```

  - **BLAS:** High-level API
    - Generic across all matrix element types: real(32|64), complex(64|128)
    - Arguments with obvious defaults are made optional

```
proc gemm(A: [?Adom] ?t, … opA = Op.N, …)
```

- **BLAS 3 (matrix-matrix) routines supported in Chapel 1.14**

BLAS Improvements: This Effort and Impact

This Effort: Added BLAS 1 & 2 support, improved interface

- BLAS 1: scalar-vector
- BLAS 2: vector-vector
  - With the exception of sparse formats: packed and banded arrays
- Dropped ldA argument from high-level interface
  - Inferred from array meta-data

Impact: BLAS module closer to completion

- Nearly full BLAS routine coverage

BLAS Improvements: Next Steps

- **100% BLAS routine coverage**
  - Support packed and banded arrays in BLAS 2
- **Explore distributed and GPU BLAS support**
  - PBLAS
  - CuBLAS, clBLAS
- **Consider Distributing a BLAS implementation with Chapel**
  - Provide out-of-the-box high performance linear algebra
  - Optionally downloaded and built as part of Chapel installation
  - BLAS can be painful for users to build depending on system

FFTW Improvements
(contributed by Nikhil Padmanabhan)

FFTW Improvements

Background: FFTW module hard-coded 'require' statements

- In FFTW.chpl:

```
require "fftw3.h", "-lfftw3";
```

- This did not support FFTW from Intel's Math Kernel Libraries (MKL)
  - MKL requires additional headers and different '`-l`' flags

This Effort: Added support for MKL implementations:

- Support MKL implementation based on 'config param':

```
chpl -s isFFTW_MKL=true fftwProgram.chpl
```

- Use new 'require' capabilities to conditionally require MKL headers
- Remove '`-l`' flags from 'require' statements

Impact: FFTW module is more flexible

Next Steps: Propagate this approach to other libraries

- BLAS and LAPACK
- Use 'config param' to distinguish FFTW from FFTW_MT

MPI Improvements

MPI Improvements

Background: MPI module supports MPI calls within Chapel

- Still a work-in-progress module
- '--spmd' flag required to specify SPMD ranks for mpirun launcher
  - Complicated testing setup for MPI SPMD mode

This Effort: Improved launcher support

- mpirun launcher given default value: '--spmd=1’
- Fixed a bug revealed by testing

Status: MPI module now tested nightly

- Run linux64 SPMD tests nightly for '--spmd=1 and '–spmd=4'

Next Steps: Improve supported configurations and features

- Support gasnet+aries, ugni, qthreads
- Add MPI-2 and MPI-3 routines

Other Library Improvements

Other Library Improvements

- RandomStream argument improvements for initializer
- ‘barrier’ changed from class to record
- conjg() now preserves type
- ‘List’ now cleans up its memory
  - contributed by Sagar Khatri
- MatrixMarket naming and bug fix improvements
- removed deprecated 'Sort' and 'Search' functions
- removed deprecated 'BigInt' class in favor of 'bigint' value

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