Standard Library Improvements

Chapel Team, Cray Inc.

Chapel version 1.12

October 1st, 2015

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.

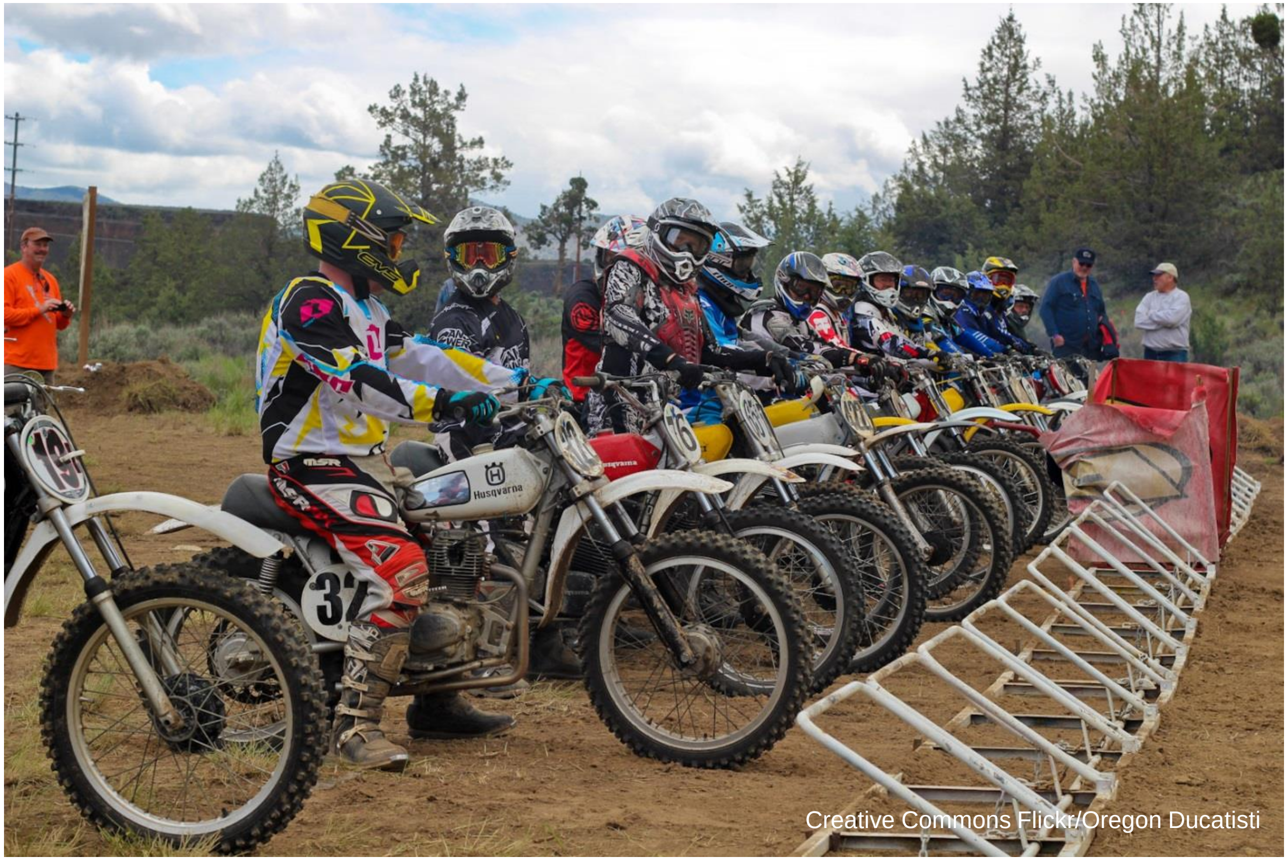
Outline

- ‘Barrier’ module: Task Barrier Synchronization
- ‘LAPACK’ Standard Module
- ‘Spawn’ module: Spawning Subprocesses
- Vectorizing Iterator
- New Math Constants in the Math module
- Other Standard Library Improvements

‘Barrier’ module: Task Barrier Synchronization

Task Barrier: Background

- All tasks must complete first step before moving on



Creative Commons Flickr/Oregon Ducatisti

Task Barrier: This Effort

- **Provide a task barrier type**
  - Prevent *k* tasks from continuing until all have notified the barrier
  - Implemented as a class in the Chapel standard library

- **Choose implementation details in constructor call**
  - e.g., two underlying counter representations
    - Atomic-based task count (the default)

      ```
      var b_atomic = new Barrier(nTasks);    // atomic task counter
      ```
    - Sync-based task count

      ```
      var b_sync = new Barrier(nTasks, BarrierType.Sync); // sync counter
      ```

- **Single-phase or split-phase, based on methods called**

Single-phase

```
b.barrier();
```

Split-phase

```
b.notify();
if b.try() then …
b.wait();
```

Task Barrier: Single-Phase Example

- **All tasks must reach the barrier before any can pass it**
  - All “entering the barrier” messages print.
  - Then all “past the barrier” messages print.

```
use Barrier;

config const numTasks = here.maxTaskPar;
var b = new Barrier(numTasks);

coforall tid in 1..numTasks {
  writeln(“Task “, tid, “ entering the barrier”);
  b.barrier();
  writeln(“Task “, tid, “ past the barrier”);
}
delete b;
```

Task Barrier: Split-Phase Example

- Tasks can do background work after notifying the barrer

```
use Barrier;

config const numTasks = here.maxTaskPar;
var b = new Barrier(numTasks);

coforall tid in 1..numTasks {
  updateSharedState();
  b.notify();
  doBackgroundWork();
  b.wait();
  readSharedState();
}
delete b;
```

Task Barrier: Distributed Example

- Barriers can synchronize tasks spanning multiple locales

```
use Barrier;
var b = new Barrier(numLocales);

coforall locid in 0..#numLocales {
  on Locales[locid] {
    writeln("Hello from locale ", here.id);
    b.barrier();
    writeln("Goodbye from locale ", here.id);
  }
}
delete b;
```

Task Barrier: Status and Next Steps

Status:

- Barrier class type functionally correct for…
  - Atomic- or sync-based counters
  - Single- or split-phase usage
  - Single- or multi-locale scenarios
- Documentation at:
  - http://chapel.cray.com/docs/1.12/modules/standard/Barrier.html

Next Steps:

- Switch to record type for automatic memory management
  - waiting on fixes to memory management for records
  - this will remove the need to delete barrier objects as in current use
- Optimize the implementation of barriers
  - particularly for the multi-locale case
- Create “task-teams” concept, and implement barriers for them

‘LAPACK’ Standard Module

LAPACK: Background and This Effort

Background:

- Users are increasingly interested in numerical libraries out of the box
  - e.g., FFTW, BLAS, GSL, **LAPACK,** etc.
- Chapel support has historically been thin
- Belief that Chapel features should result in nice interfaces
  - particularly domains/arrays and generics

LAPACK: Background and This Effort

This Effort:

- Developed by Ian Bertolacci (summer intern, Colorado State Univ.)
- Adds support for (most) LAPACK routines in two forms:
  - Idiomatic Chapel interface
  - Classic interface
- Requires users to have their own LAPACK installation
- LAPACK module generated automatically by scraping sources/docs
  - virtually necessary due to the large size of the API
- Documented online:
  http://chapel.cray.com/docs/1.12/modules/standard/LAPACK.html
- Primer example available in examples/ directory
  https://github.com/chapel-lang/chapel/blob/master/test/release/examples/primers/LAPACKlib.chpl

LAPACK: Impact

Example using idiomatic Chapel interface:

- Uses information stored in Chapel's arrays
- Benefits from Chapel's support for generic functions

```
use LAPACK, Random;

// Solve for X in A*X = B
var A : [1..5, 1..5] real;
var B : [1..5, 1..3] real;
var ipiv : [1..N] c_int; // output array of pivot indices

fillRandom(A);
fillRandom(B);

var WorkA = A;   // LAPACK will use array data as a workspace, so
var WorkBX = B;  // make copies to preserve the original data
var err = gesv(lapack_memory_order.row_major, WorkA, ipiv, WorkBX);

if err == 0 then writeln("X = ", WorkBX);
```

LAPACK: Impact

Comparing idiomatic vs. classic interfaces:

```
// idiomatic interface
var err = gesv(lapack_memory_order.row_major, WorkA, ipiv, WorkBX);

// classic interface
var err = LAPACKE_sgesv(lapack_memory_order.row_major,
                        WorkA.domain.dim(1).size:c_int,
                        WorkB.domain.dim(2).size:c_int,
                        WorkA,
                        WorkA.domain.dim(2).size:c_int,
                        ipiv, WorkBX,
                        WorkB.domain.dim(2).size:c_int);
```

Array element type embedded in routine name

Need to pass in array sizes explicitly

LAPACK: Next Steps

- **Get user feedback**
  - Does the Chapel interface make sense? Could it be better?
  - Does the classic interface add value?

- **Fix a few known gaps stemming from C interoperability**
  - Move away from using c_int's in Chapel idiomatic interfaces
  - Improve support for using C99 complex types
  - Support passing of Chapel functions to externs
  - Better handling of enums

- **Investigate ways to test/validate module**
  - Thousands of functions!

LAPACK: Next Steps

- **Improve support for row- vs. column-major order**
  - Add flags to domain maps to support either layout
  - Have idiomatic routines query this information from arrays

- **Explore post-LAPACK linear algebra support**
  - ‘LAPACK’ module is exactly that – a module wrapping LAPACK
  - Many users simply want “linear algebra”
    - would a different backing library be preferable?
      - e.g., one that is parallel and/or distributed?
      - e.g., Eigen, Trilinos, PetSc, Elemental/FLAME, PLASMA/MAGMA, …
    - support a common L.A. library with multiple backing implementations?

- **Continue adding support for other numerical libraries**
  - BLAS and subsets of GSL are next priorities

‘Spawn’ module: Spawning Subprocesses

Spawn Module: Background

- **Want Chapel to be useful for multi-program composition**
  - would like it to be viable as a parallel scripting language
  - e.g., for all files in this directory, compress them with `gzip`
- **Chapel did not support spawning other processes directly**
  - was possible through the extern interface, but awkward

Spawn Module: This Effort

- **Add a new 'Spawn' standard module**
  - inspired by Python's Subprocess module
  - input and output are available for redirection
  - C runtime supports this with `posix_spawn`

```
use Spawn;

// create a subprocess running md5sum
var sub = spawn(["md5sum", filename], stdout=PIPE);

// consume each line from the spawned md5sum process
var line:string;
while sub.stdout.readline(line) {
  write("md5sum returned: ", line);
}

// perform any remaining communication and wait for process to exit
sub.communicate();
```

Spawn Module: Impact

- **Enhances Chapel's ability to compose multiple programs**
  - Improves support for parallel scripting workflows
  - Supports multi-language integration in a coarse-grained manner
  - Enables program re-use
- **Used to improve a Twitter processing benchmark**
  - Benchmark wanted to read `gzip` compressed files
    - Previously, were unzipping the files manually…
    - This was our motivation for implementing the Spawn module
    - Now the program spawns `gzip` commands to read the files

Spawn Module: Status and Next Steps

Status:

- Spawn feature implemented and documented
  - http://chapel.cray.com/docs/1.12/modules/standard/Spawn.html
- Problems with `CHPL_COMM=ugni` when redirecting input or output
  - Observed seg faults from `clone()` system call
  - Added `halt()` for a better error message for this known issue
  - `CHPL_COMM=gasnet` with `aries` conduit works

Next Steps:

- Improve support for `CHPL_COMM=ugni`
- Add other ways of providing input and output:
  - file path
  - a Chapel file
  - a Chapel channel
- Fill in other missing functionality
  - continue to draw on Python's Subprocess for inspiration
- Get feedback from users

Vectorizing Iterator

Vectorizing Iterator: Background

- **Vectorization is crucial for achieving peak performance**
  - True for commodity and HPC systems
  - Becoming increasingly important, particularly in HPC
    - AVX-512 (Xeon and Xeon Phi)
    - NEON (ARM)

- **Previous releases focused on "implicit" vectorization**
  - Generating idioms the back-end can better auto-vectorize
  - Emitting hints to the back-end for vectorizable Chapel constructs
    - i.e. foralls, promoted expressions, etc
  - However, no way vectorize without also creating tasks
    - desirable for loops with small trip counts

Vectorizing Iterator: This Effort

- **Provide a simple way to vectorize without task creation**
  - Implemented as a “wrapper” iterator
    - e.g. to vectorize range iteration:

```
for i in 1..10 {…}   =>   for i in vectorizeOnly(1..10) {…}
```

- **Asserts order-independence and disables task creation**
  - i.e. same result when invoked with a serial or data parallel-loop

for example:

```
forall i in vectorizeOnly(1..10) {…}
for    i in vectorizeOnly(1..10) {…}
```

both effectively generate:

```
#pragma ivdep
for (i=0; i<=10; i+=1) {…}
```

Vectorizing Iterator: This Effort

- **Automatically handles zippering**

  to vectorize:

  ```
  for (a, b) in zip(A, B) {…}
  ```

  simply write:

  ```
  for (a, b) in vectorizeOnly(A, B) {…}
  ```

Vectorizing Iterator: Status and Next Steps

Status:

- vectorizeOnly() iterator implemented
  - has significant correctness testing
  - further performance evaluation is required

Next Steps:

- Evaluate performance impact of vectorizeOnly() using LCALS
- vectorizeOnly() clean-up:
  - move into a standard module (will not be implicitly included)
  - improve orthogonality with zippering
  - consider generating warning for serial (for-loop) invocations
- Create a vectorization primer as a guide for:
  - implicit vectorization
  - vectorizeOnly() iterator

New Math Constants in the Math module

Math Constants Added to the Math module

Background: Math.chpl was missing constants for *pi*, *e*, etc

- These are in C's math.h, but were omitted from Chapel's Math.chpl

This Effort: Adds math constants found in C's math.h

param `pi` = 3.14159

pi - the circumference/the diameter of a circle

param `half_pi` = 1.5708

pi/2

Next Steps: Consider compile-time param real evaluation

- Constants like **half_pi** might not be necessary
- Could/should compiler evaluate **param real** expressions like **pi**/2 ?

Other Standard Library Improvements

Other Standard Library Improvements

- **renamed memory diagnostics symbols for clarity**
- **improved format() routine**
  - changed from standalone function to string.format()
  - unified format strings with writef()
- **in format strings, “##.##” now requires curly brackets**
  - old scheme led to challenges, e.g. when wanting to print ‘#’ characters
- **added getFileSize() to ‘FileSystem’ module**
- **‘UtilMath’ module merged into ‘Math’**
- **errorToString() now portably returns ‘No Error’**
- **applied ‘private’ to standard module symbols that are**
  - a few cases remain due to tests that refer to them, needing rewriting

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, URIKA, and YARCDATA. 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.

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/