Ongoing Efforts

Chapel Team, Cray Inc.
Chapel version 1.14
October 6, 2016

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

- **Compiler**
  - Improving Array Memory Management
  - Error Handling
  - Stack Allocate Argument Bundles
  - Partial Reductions
- **Modules**
  - Chapel on Intel Xeon Phi “Knights Landing” (KNL)
  - DefaultRectangular Multi-DData
  - Mason: A Package Manager for Chapel
- **Applications**
  - Machine Learning

Ongoing Compiler Efforts

Improving Array Memory Management

Arrays: Outline

- **What is the problem?**
  - Array memory management was incorrect and slow
- **Why do we have this problem?**
  - Original semantics of arrays required reference counting
- **How did we address the problem?**
  - Language changes
  - Leveraging improved semantics
- **What is the result of this work?**
  - Huge reduction in leaks, varied performance impact

Arrays: Background: The Problem

- **Array memory management has been problematic**
  - memory leaks
  - performance overhead
- **Largest source of memory leaks**
  - distributed array leaks account for most leaked data
  - privatized objects are leaked
  - distributions, distributed domains leak as well
- **Significant overhead reduces performance**
  - Array memory management overheads can be surprising:

    ```
    var size = A.domain.size; // changes reference counts!
    ```
  - Benchmarks spend significant time handling array reference counting
    - Have supported a 'noRefCount' setting to measure/reduce impact
    - Sometimes dramatic, but guaranteed arrays will be leaked

Arrays: Background: How did we get here?

- **Arrays are implemented to keep arrays alive**
  - when an array slice outlives the original array
  - when arrays are used in begin statements

```
proc run() {
  var A:[1..100] int;
  begin {
    computeWith(A);
  }
} // local variables normally destroyed here
```

- **And at the same time, to minimize array copies**
- **But…**
  - Implementation erred on keeping arrays alive to the point of leaking
  - Reference counting approach was expensive and overly conservative
  - Language definition did not clearly specify array return behavior

Arrays: This Effort

- **To solve these problems we**
  - altered the behavior of returning arrays
  - leveraged this change to eliminate reference counting
- **Lexical scoping eliminates need for reference counting**
  - arrays should be freed when they go out of scope
  - begin statements need not prevent arrays from being freed
  - array slices need not prevent arrays from being freed
- **Re-implemented array, domain, distribution types to:**
  - remove array reference counting
  - free distributed objects
  - rely on fewer special cases in the compiler

Array Challenges: Tuple Improvements

- **Tuple semantics have never been well-defined**
  - a known gap in the language specification
  - CHIP-6 proposed one strategy, but was never finalized or acted upon
  - things have worked "well enough" for this not to receive more attention
- **The array effort ran afoul of issues with tuples**
- **Led us to rework the tuple implementation**
  - Guiding principal: 1-element tuples behave similarly to plain elements
  - implementation is now more direct and straightforward

Array Challenges: Reference Types

- **Compiler has longstanding issues with representing 'ref's**
  - inconsistent representations
  - incorrectly identified
- **The array effort ran into challenges related to this**
- **Motivated a new approach: separate ref-ness from type**
  - new 'Qualifier' IR component
    - ref, const-ref, value, param, unknown, …
    - references now correctly identified, no longer in the type
- **Still need to propagate this change through the compiler**
  - working our way backwards through the passes
- **A positive change independent of arrays work**
  - fewer record copies in some cases
  - addresses long-standing pain-point for compiler developers

Arrays: This Effort: Language Changes

- **Arrays are now destroyed when they go out of scope**
  - begin statements and array slices no longer affect array lifetime
- **Arrays return by value by default**
  - ref and const ref return intent request alternative behavior

Arrays: Now Destroyed When Out Of Scope

- **Using arrays past their scope is now a user error:**

```
proc badBegin() {
  var A: [1..10000] int;
  begin {
    A += 1;
  }
  // Error: A destroyed here at function end, but the begin could still be using it!
}
```

- **Using a slice after original array destroyed now an error:**

```
proc badSlice() {
  var A: [1..10000] int;
  var slice => A[1..1000];
  return slice;
  // A destroyed here at end of the function, but the returned slice refers to it!
}
```

Arrays: Now Return by Value By Default

- **Now the act of returning an array makes a copy:**

```
var A: [1..4] int;
proc f() {
  return A;
}
ref B = f();
B = 1;
writeln(a);
// outputs 1 1 1 1 historically
// outputs 0 0 0 0 after this work
```

- **Old behavior available with ref return intent:**

```
proc f() ref {
  return A;
}
```

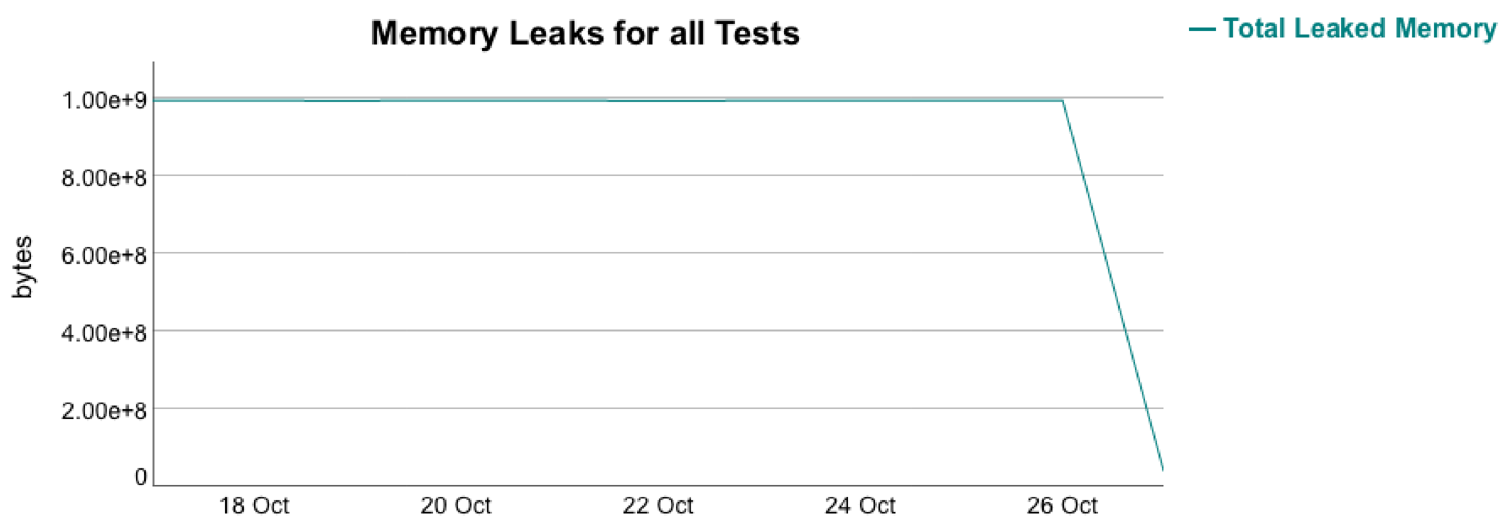
Arrays: Status

Status:

- Array improvements merged to master on Oct 26th
  - testing reasonably clean, particularly given magnitude of changes
  - graphs on following slides are taken from the next day's results
- Had hoped to include this in 1.14, but did not feel sufficiently confident
  - it's a significant change
  - even if it could have been ready in time, wanted to live with it for awhile
- Closes biggest memory leak sources
- Generally improves performance

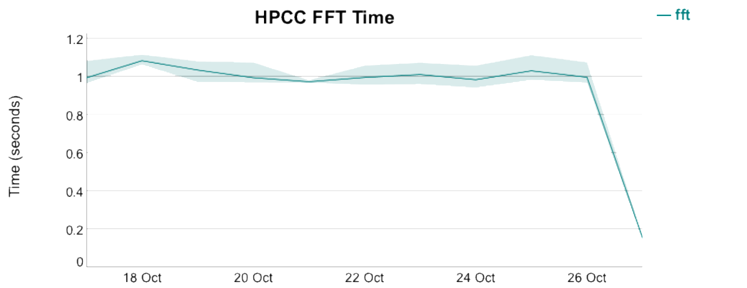
Arrays: Impact on Leaks

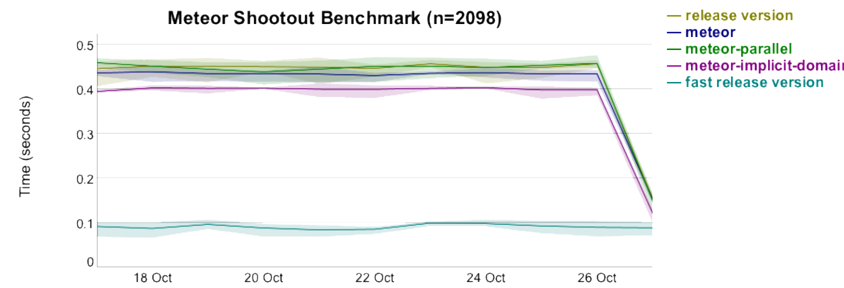
- **Memory leaks were dramatically reduced**
  - Biggest source of memory leaks closed
    - PTRANS benchmark went from leaking 800MB to 0 bytes
  - Distributed arrays no longer leak
    - With the exception of ReplicatedDist (default constructor issue)

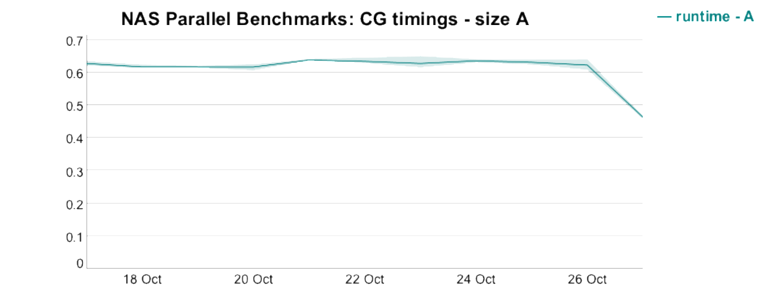


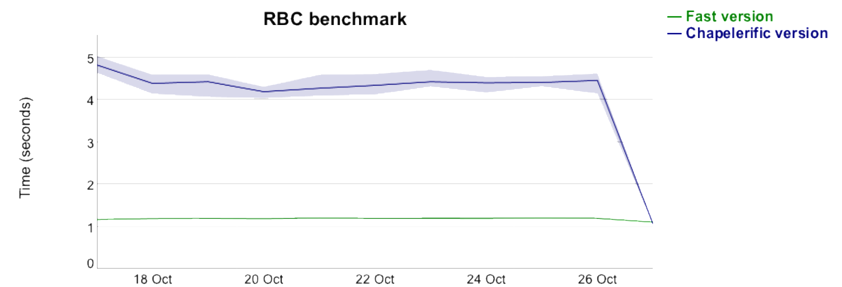
Arrays: Impact on Performance

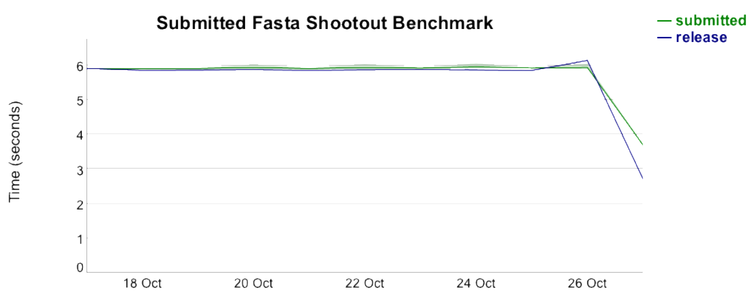
- Substantial single-locale performance improvements

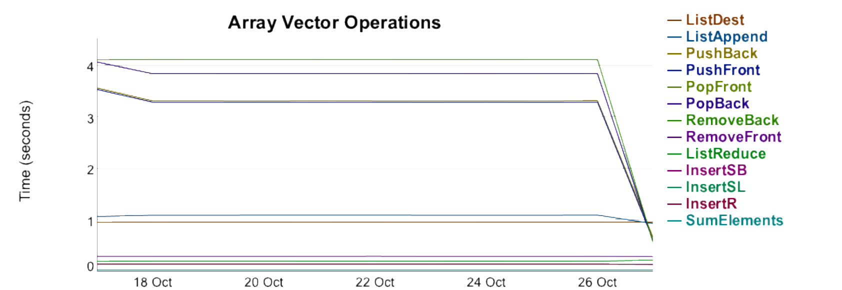






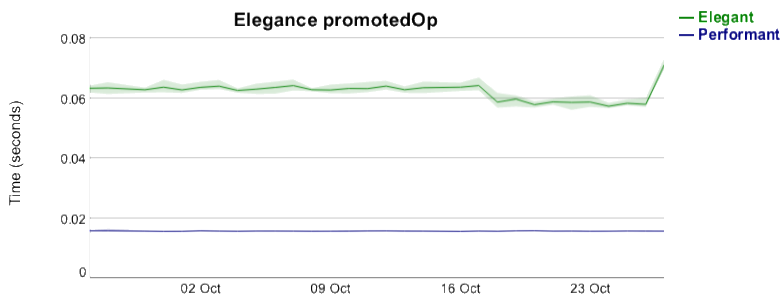




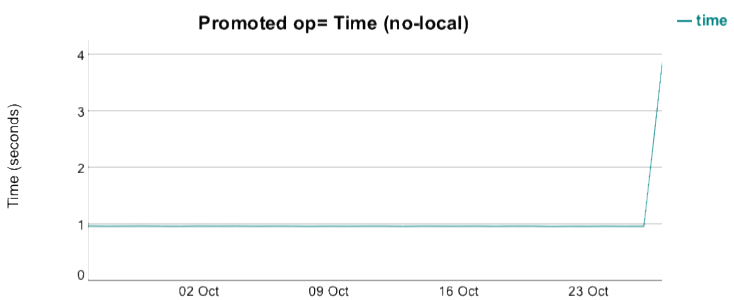


Arrays: Impact on Performance

- **Some minor single-locale performance regressions**

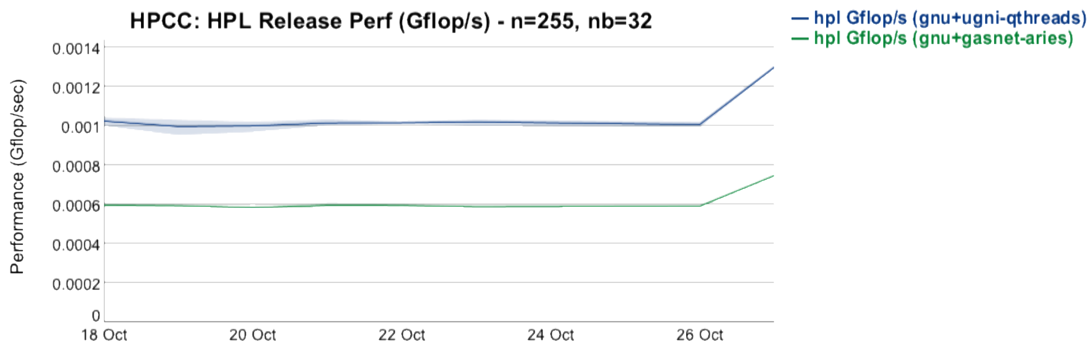


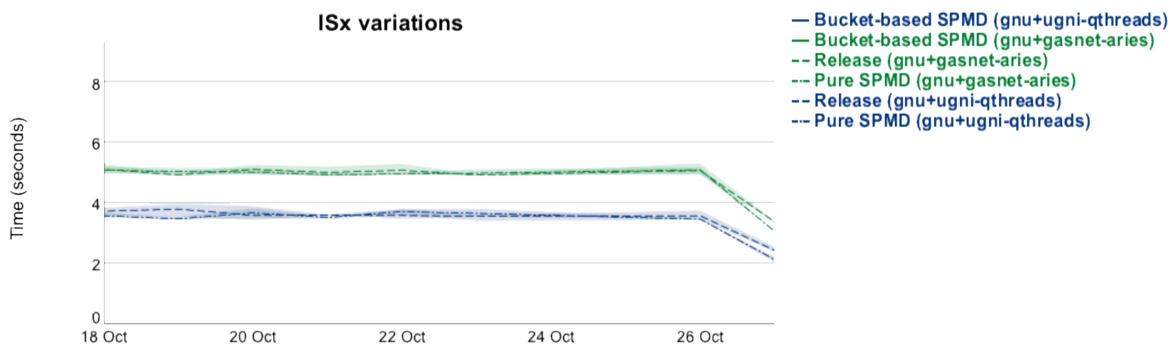
- **Only one major single-locale performance regression**
  - Have not had time to investigate yet



Arrays: Impact on Performance

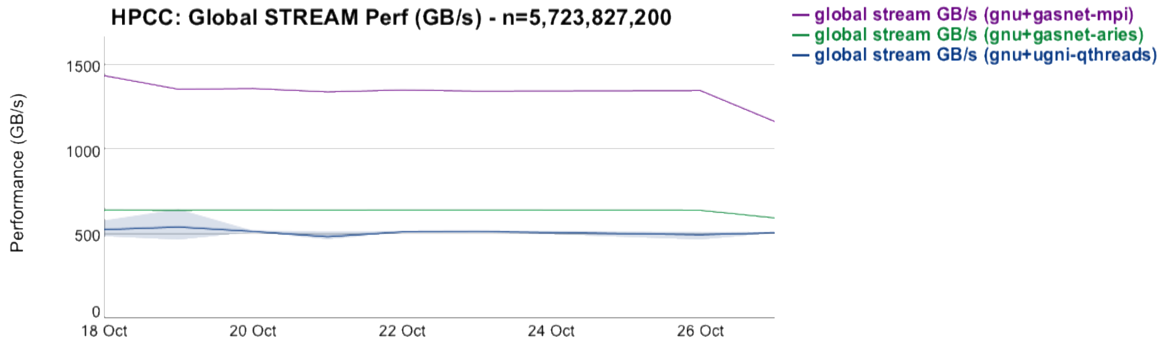
- Some decent multi-locale performance improvements

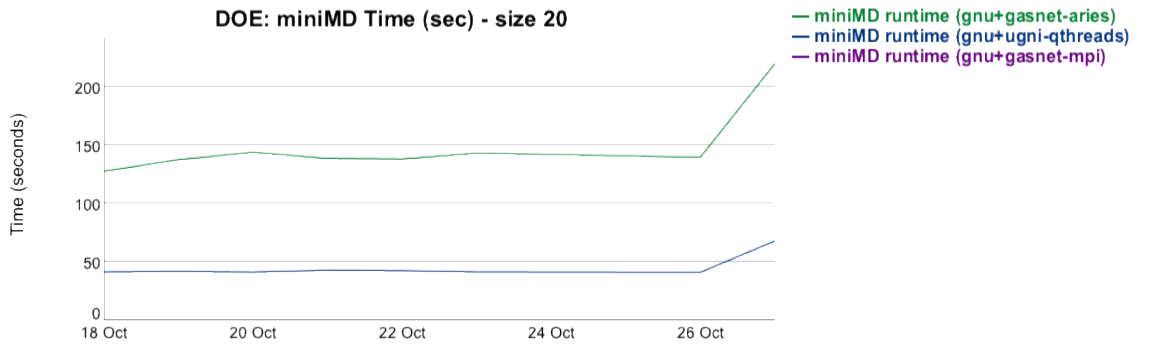




Arrays: Impact on Performance

- **A few multi-locale performance regressions**
  - Root cause still needs to be investigated





Arrays: Next Steps

Next Steps:

- Investigate and resolve performance regressions
  - not surprising that there were a few regressions
  - code was just merged, need some time to investigate
- Close remaining memory leaks
  - issue with generic member variables causing leak with ReplicatedDist
  - undiagnosed leak observed in some applications
  - close non-array-based memory leaks
- Optimize away deep array copies when possible
  - e.g., returning local arrays and assigning result
  - extend this optimization to other record types as well (e.g., ‘bigint’)
- Update language specification to describe:
  - when record/array copies occur
  - tuple semantics in more detail
  - function return is normally by-value

Error Handling

Error Handling: Background

- **Chapel lacks a general strategy for handling errors**
  - Current approaches: `halt()` and "error" out arguments
  - These are insufficient moving forward
- **First cut at an improved design was modeled after Swift**

```
proc canThrowErrors() throws { ... }
do {
  try canThrowErrors();
  try! canThrowErrors(); // will halt on failure
} catch {
  writeln("first call failed!");
}
```

  - All calls that might throw must be marked with `try`
    - Makes control flow clear with only local information
    - Feedback was that this is too verbose
  - Early feedback from users, developers wanted a way to elide try
    - e.g., lack of try results in halt-like behavior

Error Handling: This Effort

- **Improve the syntax and semantics**
  - Streamline for usability
  - Maintain clarity of control flow
- **Decide on an implementation approach**
- **Design document can be found in CHIP #8**

Error Handling: syntax and semantics

- **Mark error-throwing procedures with `throws`**
- **Throwing calls must be enclosed by `try` or `try!`**
  - Eliminated `do`
  - Defined for single statement and { } blocks
  - Both will attempt to match errors to a `catch` block
- **If a matching `catch` is not found:**
  - `try` propagates the error
    - To an outer `try`, or out of the procedure (which must throw)
  - `try!` is similar but halts instead of propagating

```
try {
  canThrowErrors();          // handled by catch on error
  try canThrowErrors();      // propagated to outer try, which goes to catch
  try! canThrowErrors();     // halts on error
} catch {
  writeln("first call failed!");
}
```

Error Handling: default and strict mode

- **Tension between convenience and correctness**
- **Default mode**
  - If a throwing procedure is not enclosed with `try`:
    - Halt if that procedure throws an error
- **Strict mode**
  - If a throwing procedure is not enclosed with `try`:
    - Raise a compiler error
- **To start, toggle between modes with a compiler flag**
  - Expect to include a more fine-grained approach in the future
    - Per-module or per-function

Error Handling: errors as classes

- **Long-term goal of supporting many types as errors**
  - classes, records, enums, unions, tagged unions, etc.
- **To start, all errors must be classes**
  - Implementation convenience
  - Base class `Error` will be provided
- **`catch` blocks match against an `Error` at runtime**

```
try {
  trickyOperation(badArg);
} catch IllegalArgumentError {   // matches IllegalArgumentError, subtypes
  writeln("illegal argument!");
} catch {                        // catch-all block, prevents auto-propagation
  writeln("unknown error!");
}
```

Error Handling: implementation approach

- **Compiler translates into 'out' error arguments**

```
// Example: function signatures
proc canThrowErrors() throws { ... }
// translates into
proc canThrowErrors (out _e_out) { ... }
```

```
// Example: try-throws
proc caller() throws {
  try canThrowErrors();
}
// translates into
proc caller(out _e_out) {
  var _e: Error;
  canThrowErrors(_e);
  if _e then
    _e_out = _e;
}
```

```
// Example: try!
proc handler() {
  try! canThrowErrors();
}
// translates into
proc handler() {
  var _e: Error;
  canThrowErrors(_e);
  if _e then
    halt(_e.message);
}
```

Error Handling: implementation approach

```
// Example: try-catch
proc catch() {
  try {
    canThrowErrors();
  } catch {
    writeln("error occurred");
  }
}
// translates into
proc catch() {
  var _e: Error;
  canThrowErrors(_e);
  if _e then
    writeln("error occurred");
}
```

```
// Example: try-catch-throws
proc attempt() throws {
  try {
    canThrowErrors();
  } catch e: SomeError {
    writeln(e.message);
  }
}
// translates into
proc attempt(out _e_out: Error) {
  var _e: Error;
  canThrowErrors(_e);
  if _e: SomeError then
    writeln(_e.message);
      else if _e then
    _e_out = _e;
}
```

Error Handling: runtime errors

- **C runtime is independent of Chapel error handling**
  - Modify runtime to return error codes
  - Chapel wrapper translates runtime error codes to throws

```
// Example: in module code
proc chpl_here_alloc(size: int): c_void_ptr throws {
  extern proc chpl_mem_alloc(size: int) : c_void_ptr; // runtime proc
  const p = chpl_mem_alloc(size); // always returns
  if p == c_nil then              // runtime says allocation failed
    throw new OutOfMemoryError();
  return p;
}
```

- **More involved implementation**
  - Will not be included in initial version

Error Handling: error cleanup

- **`defer`, Swift's cleanup construct**
  - Runs whenever the enclosing scope exits
  - Cleanup code is local to initializing code
  - Useful outside of error handling
- **Chapel will adopt something similar to `defer`**

```
proc caller() throws {
  try {
    var a = allocateBigObject();
    defer {
      delete a;
    }
    canThrowErrors(a);
  }
}
```

- **More involved implementation**
  - Will not be included in initial version

Error Handling: Status and Next Steps

Status:

- Design is hosted on the Github repository as CHIP 8
  - Advertised to the community and solicited feedback

Next Steps:

- Implement the design in the compiler
- Release the feature to users in Chapel 1.15.0

Stack Allocate Argument Bundles

Stack Allocate Argument Bundles

Background: on/begin/cobegin/coforall create argument bundles

- these argument bundles are heap-allocated in the generated code
- heap allocation can be a significant source of overhead

This Effort: compiler generates code to stack-allocate bundles

- the runtime can copy to a heap-allocated region when appropriate
  - in fact, the runtime already does so in many cases because the generated code might free the bundle before it is used
- runtime can identify when a bundle could be destroyed before use

Impact: Observed 10-20% speedups for LULESH and MiniMD

Next Steps: Complete the change

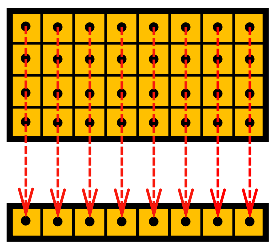
- nontrivial effort since it touches all tasking and communication layers

Partial Reductions

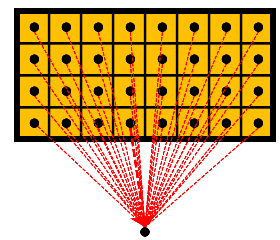
Partial Reductions: Background

- *Partial* reductions reduce over subset of array dimensions

ex. partial reduction over columns produces a row



cf. *full* reduction produces a single element



- Partial reductions are not available in Chapel at present
  - Important for many algorithms, particularly matrix-vector multiplication
  - Can be written manually, but typically at greater expense

Partial Reductions: This Effort – Goals

Design and implement partial reductions

- **Candidate syntax:**

"this is a partial reduction over 2nd dimension"

```
var Q = + reduce only(2) [(i,j) in MatDom] (A(i,j) * P(j));
```

partial reduction over 1st and 3rd dimensions produces a single column for a 3D array

```
const C = + reduce only(1,3) myCube;
```

Partial Reductions: This Effort – Prototypes

Developed templates for compiler to apply

- "Got a partial reduction? Replace it with this piece of code."
- Implemented partial + reductions of arrays, over a single dimension
- Relies on proposed support within array's domain map
  - **proc** dsiPartialDomain() → domain with remaining dimension(s)
  - **iter** dsiPartialThese() → how to iterate over remaining dimension(s)
  - domain map gets to decide which locales are involved, in what order, etc.

- **Basic template**

*// implements: result = + **reduce only**(2) myArray;*

```
forall partIdx in myArray.domain.dsiPartialDomain(exceptDim=2)
do result[partIdx] =
     + reduce myArray.dsiPartialThese(2,partIdx);
```

- **"Bulk" template**
  - communicate per-locale results in bulk to improve performance
  - interface and performance need further development

Partial Reductions: Next Steps

- **Have compiler invoke templates**
  - implement parser changes
  - convert reduction operation into templates
  - avoid creating temporary result if user copies it into an array anyway
- **Generalize to arbitrary expressions, multiple dimensions**
- **Consider bringing ZPL flood/grid dimensions to Chapel**
  - Useful in reduction scenarios for both users and implementers
- **Improve performance**
  - finalize "bulk" template

Ongoing Module Efforts

Chapel on Intel Xeon Phi “Knights Landing” (KNL)

Chapel on KNL: Background

- KNL is a many-core platform (60+ cores).
- Access both to external memory and to on-chip high-bandwidth multichannel DRAM (MCDRAM).
- Presents an opportunity to broaden Chapel’s NUMA support in preparation for more complex architectures.

Chapel on KNL: This Effort

- **We examined MCDRAM performance characteristics**
  - Experimented with real Chapel code using and not using MCDRAM
- **For many Chapel codes, MCDRAM makes little difference**
- **On two key benchmarks, saw a significant difference**
  - Streaming: ~50% speedup on tested Chapel code
  - Random-access: ~20% slowdown on tested Chapel code
- **As a result, we changed our strategy**
  - We were planning to allow users to ask for "fast" memory
  - Now we plan to split this into two kinds of "fast"
    - Streaming
    - Random Access
  - Similar to the way the *madvise()* system call works on pages

Chapel on KNL: Status and Next Steps

Status:

- Have prototype locale model from previous release cycle
- Gained better understanding of KNL performance characteristics
- Changed plans regarding mechanism of MCDRAM support

Next Steps:

- Leverage previous and new insights to enhance NUMA support
- Target KNL in a way that can be used for future architectures
  - KNL Locale Model
  - “Get streaming memory” or “get random-access memory”
- Take advantage of latest Qthreads, with better KNL support
- Work on vectorization improvements/optimizations
  - Also explore potential KNL-specific optimizations

DefaultRectangular Multi-DData

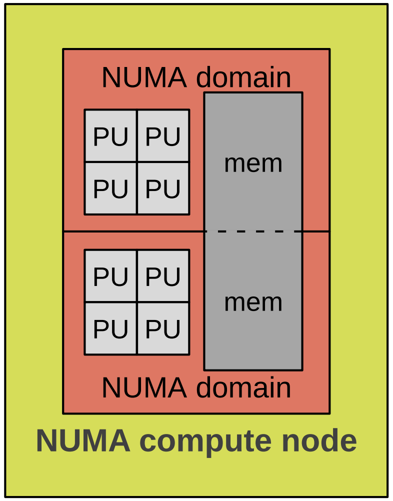
DefaultRectangular Multi-DData: Effort

- **Background:** 'numa' locale model doesn't perform as desired
  - DefaultRectangular domain and arrays lack sublocale optimizations
  - domain places tasks as desired (subject to tasking layer limitations)
  - but arrays have 1 *ddata* data block per node, not explicitly placed
    - thus: no reliable data/task affinity

- **This Effort:** Localize data as DefaultRect domain does tasks

```
forall … do <something> ;

coforall … on … {      // across numa sublocales
  coforall … on … {    // across PUs within subloc
    for … do <part of something in subloc's mem>
  }
}
```



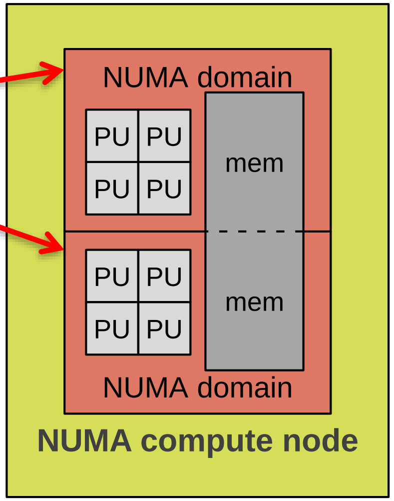
  - implement *multi-ddata* arrays: 1 ddata per sublocale
    - used whenever locales have sublocales (e.g. numa)
    - set NUMA/sublocale affinity of ddata blocks via OS

DefaultRectangular Multi-DData: Effort

- **Background:** 'numa' locale model doesn't perform as desired
  - DefaultRectangular domain and arrays lack sublocale optimizations
  - domain places tasks as desired (subject to tasking layer limitations)
  - but arrays have 1 *ddata* data block per node, not explicitly placed
    - thus: no reliable data/task affinity
- **This Effort:** Localize data as DefaultRect domain does tasks

```
forall … do <something> ;

coforall … on … {      // across numa sublocales
  coforall … on … {    // across PUs within subloc
    for … do <part of something in subloc's mem>
  }
}
```



  - implement *multi-ddata* arrays: 1 ddata per sublocale
    - used whenever locales have sublocales (e.g. numa)
    - set NUMA/sublocale affinity of ddata blocks via OS

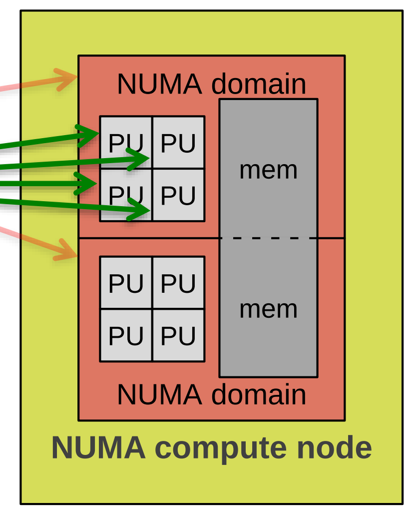
DefaultRectangular Multi-DData: Effort

- **Background:** 'numa' locale model doesn't perform as desired
  - DefaultRectangular domain and arrays lack sublocale optimizations
  - domain places tasks as desired (subject to tasking layer limitations)
  - but arrays have 1 *ddata* data block per node, not explicitly placed
    - thus: no reliable data/task affinity

- **This Effort:** Localize data as DefaultRect domain does tasks

```
forall … do <something> ;

coforall … on … {      // across numa sublocales
  coforall … on … {  // across PUs within subloc
    for … do <part of something in subloc's mem>
  }
}
```



  - implement *multi-ddata* arrays: 1 ddata per sublocale
    - used whenever locales have sublocales (e.g. numa)
    - set NUMA/sublocale affinity of ddata blocks via OS

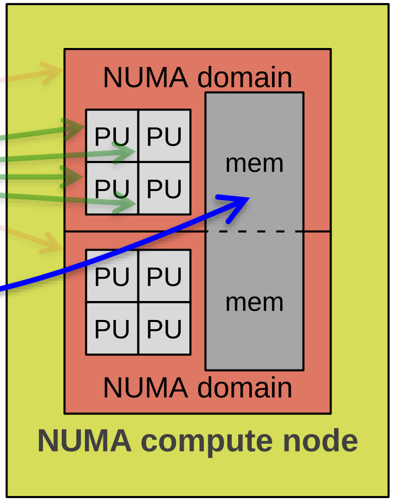
DefaultRectangular Multi-DData: Effort

- **Background:** ‘numa’ locale model doesn’t perform as desired
  - DefaultRectangular domain and arrays lack sublocale optimizations
  - domain places tasks as desired (subject to tasking layer limitations)
  - but arrays have 1 *ddata* data block per node, not explicitly placed
    - thus: no reliable data/task affinity

- **This Effort:** Localize data as DefaultRect domain does tasks

```
forall … do <something> ;

coforall … on … {      // across numa sublocales
  coforall … on … {  // across PUs within subloc
    for … do <part of something in subloc’s mem>
  }
}
```



  - implement *multi-ddata* arrays: 1 ddata per sublocale
    - used whenever locales have sublocales (e.g. numa)
    - set NUMA/sublocale affinity of ddata blocks via OS

DefaultRectangular Multi-DData: Status

History & Status:

- did not make as much progress as hoped and planned this cycle
- currently working on more principled array index → ddata mapping
  - old: leftmost dim with range >= #sublocales, must divide evenly
  - new: same, but without "divide evenly", ±1 index val per ddata block
    (exactly matches task distribution)
- array remote-access data (RAD) opt, bulk I/O, bulk transfer not done

Next Steps:

- finish new index → ddata mapping
- finish RADopt, bulk I/O, bulk transfer
- runtime tasking: implement task placement in qthreads tasking layer
- memory management:
  - numa-awareness in runtime memory layer(s)
  - on Cray X* systems, integrate with use of hugepages

Mason: A Package Manager for Chapel

Mason: Background

- **Currently packages are bundled with the rest of Chapel**
- **A lot of drawbacks and few benefits:**
  - Developers must sign a CLA
  - Code must be under a compatible license
  - Chapel core team needs to be involved
  - Packages must serve a wide audience
  - Bound to Chapel's six month release cycle
- **Not ideal for a healthy Chapel ecosystem**

Mason: This Effort

- **Designed a package management system for Chapel**
  - Package manager
  - Build system
  - Package registry
- **Trying to avoid reinventing the wheel**
  - Shares traits with Rust's Cargo and homebrew
- **Design document can be found in [CHIP #9]**

Mason: Sample Workflow

- **Mason is a command line tool for package management**
  - Also manages project builds
- **Getting started**
  - Initialize the project directory

    ```
    mason new [project name] ...
    ```
  - For project name foo, this produces:

    ```
    Foo/
      Mason.toml
      src/
        Foo.chpl
    ```
  - Write your project code
  - Build your project

    ```
    mason build
    ```
    - In our example, this will compile `Foo.chpl`

Mason: Sample Workflow

- **Dependency management**
  - Add or remove dependencies

    `mason add [package] [version]`

    `mason rm [package]`
  - Pulled in and included by `mason build`
  - Dependency code is downloaded to a common pool of packages
- **Project manifest file**
  - `Mason.toml`
  - Tracks dependencies
    - Edited automatically by mason
    - May be edited manually
  - Stores project metadata
    - Must be edited manually (name, version, authors, license, etc.)

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Bradford Chamberlain <brad@chamberlain.com>"]
license = "Apache-2.0"

[dependencies]
Curl = "1.0.0"
```

Mason: Package Registry

- **Implementation**
  - Github repository of package manifest files
  - Identical to the one in the project, plus a source url field
  - Publish a package by submitting a pull request
- **Issues**
  - Namespacing
    - First-come, first-served
  - Versioning
    - Semantic versioning
  - Integrity
    - Travis CI suite
    - Review board
  - Licensing
    - SPDX

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Brad Chamberlain <brad@chamberlain.com>"]
license = "Apache-2.0"
source = { git = "https://github.com/bradcray/hello_world", tag = "0.1.0" }

[dependencies]
Curl = "1.0.0"
```

Mason: Implementation Details

- **Lock file**
  - `Mason.lock`
  - "Locks in" a build configuration from the manifest
    - Serialized DAG of all dependencies
    - Points to specific Git SHAs
  - Ensures repeatable builds on other machines
  - After editing a manifest, generate a new lock
    - `mason update`
- **Syncing commands**
  - mason is a pipeline
    - source → manifest → lock → dependency code
  - When mason commands are run, keep them in sync
    - ex. `mason add`
    - triggers `mason update`, downloaded dependencies

Mason: Status and Next Steps

Status:

- Design is hosted on the Github repository as CHIP 9
  - Advertised to the community and asked for feedback

Next Steps:

- Implement `mason` in Chapel
- Build and release the first version

Ongoing Application Efforts

Machine Learning

Machine Learning: Background

- **Deep learning frameworks have various limitations/issues**
  - Frameworks support narrow range of parallel & memory architectures
    - Limited by the choice of parallel libraries used in implementation
  - Supporting more architectures requires combining many libraries
    - All with their own syntax and semantics
    - Introduces complexity and burdens framework development
  - Typically written in C/C++
    - Often requiring interface with more productive languages, like Python or R

Machine Learning: Background

- **Chapel provides solutions to current issues**
  - Natively supports a wide range of parallel & memory architectures
    - The list continues to grow
  - Parallelism expressed under the same syntax and semantics
  - Chapel is productive and performant
    - And can still provide interoperate with other languages

- **Machine learning is a good use-case for Chapel analytics**
  - Deep learning identified as good area to focus within machine learning
    - Incredible amount of traction in research & industry
    - Often requires significant computational resources
    - Consequently, scalable implementations are necessary

Machine Learning: This Effort

- **Built some toy ML codes**
  - Helped develop an understanding of the basics
  - Helped determine what building blocks are needed in Chapel for ML

- **Identified necessity for high-level linear algebra interface**

- **Created a linear algebra module: LinearAlgebra.chpl**
  - Provides high-level syntax for linear algebra routines
    - Similar in nature to Python's NumPy or Matlab's linear algebra interface
  - Built on top of BLAS / LAPACK
    - Users can swap out implementations as needed
    - Could some day be entirely Chapel, as performance permits

Machine Learning: Impact

- **Examples of LinearAlgebra interface:**
  - Matrix-matrix / matrix-vector multiplication: `dot(A, B)`
  - Matrix transpose: `A.T()`
  - Matrix inverse: `inverse(A)`
  - Identity matrix: `eye({1..10, 1..10})`

- **Some toy ML codes built:**
  - Normal Equation
  - Gradient Descent
  - Neural network
    - Creates an N-layer network with any number of neurons per layer
    - Supports stochastic gradient descent with backpropagation

Machine Learning: Status and Next Steps

Status: Progress towards machine learning in Chapel

Next Steps: Continue to explore ML in Chapel

- Publish Linear Algebra module in 1.15
  - Add additional routines
  - Documentation
  - Integrate BLAS/LAPACK installation with Chapel build system
- Combine existing toy codes into a simple ML framework
- Ramp up toy code implementations, and compare to other frameworks
  - Explore optimizations and parallel implementations
- Work towards a deep learning application
  - Identify or create a benchmark to measure Chapel's performance
  - Collaborate with researchers on real-world applications

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