Language 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

- Core Language Improvements
- Improvements to Intents
- Improvements for Generics
- Other Language Improvements

Core Language Improvements

Core Language Improvements

- Method Forwarding
- Void Fields
- ‘Require’ Improvements
- Module Deinit

Method Forwarding

Forwarding: Background

- **As with other languages with classes…**

  ... Chapel has had two ways of reusing methods:
  1. reuse by inheritance
  2. reuse by composition

- **Inheritance is not always appropriate**
  - it affects how the inherited object can be used
    - e.g. can cast to the parent class
    - inheritance creates an "*is a*" relationship
  - class authors may not wish to support inheritance
    - challenging to create a good API that expects inheritance
  - see also "composition over inheritance" principle / design pattern

- **Chapel doesn't support some inheritance patterns:**
  - multiple inheritance
  - record inheritance

Forwarding: Motivation

- **Suppose we have MyCircle and it wraps MyCircleImpl**
  - MyCircle is a record we intend to provide to users
  - MyCircleImpl is a class that implements the methods

```
class MyCircleImpl {
  var radius: real;


  proc area() {
    return pi*radius*radius;
  }
  proc circumference() {
    return 2.0*pi*radius;
  }
}
```

```
record MyCircle {
  var impl: MyCircleImpl;

  // forwarding methods
  proc area() {
    return impl.area();
  }
  proc circumference() {
    return impl.circumference();
  }
}
```

- **Such record wrapper patterns are common in Chapel**
  - Writing such forwarding methods can be cumbersome
    - especially for generic wrapper types such as 'Owned' and 'Shared'

Forwarding: This Effort

- **Add a 'forwarding' feature for field declarations**
  - Supports auto-forwarding of unresolved methods to that field
  - Previous example can now be written as:

```
class MyCircleImpl {
  var radius: real;


  proc area() {
    return pi*radius*radius;
  }
  proc circumference() {
    return 2.0*pi*radius;
  }
}
```

```
record MyCircle {
  forwarding var impl: MyCircleImpl;
  // above declaration requests forwarding

  // compiler creates area() method
  // to call impl.area()

  // compiler creates circumference() method
  // to call impl.circumference()

}
```

Forwarding: This Effort

- **Add a 'forwarding' feature for field declarations**
  - Note that methods handled by the original object are not forwarded:

```
class MyCircleImpl {
  var radius: real;


  proc area() {
    return pi*radius*radius;
  }
  proc circumference() {
    return 2.0*pi*radius;
  }
  proc whoAmI() {
    writeln("class");
  }
}
```

```
record MyCircle {
  forwarding var impl: MyCircleImpl;
  // above declaration requests forwarding

  // compiler creates area() method
  // to call impl.area()

  // compiler creates circumference() method
  // to call impl.circumference()

  proc whoAmI() {
    writeln("record");
  }
}
```

Forwarding: More Details

- **'forwarding' declarations…**
  …indicate where to forward otherwise unresolved method calls
  …can be used multiple times inside a class or record declaration

- **Two styles of use:**
  - as a field declaration prefix:
    ```
    forwarding var myField;
    ```
  - as a standalone member declaration that refers to a field:
    ```
    var myField;
    forwarding myField;
    ```

- **Filter forwarded methods with 'only' and 'except' lists**
  - similar to ‘only’ and ‘except’ on module ‘use’ statements
    ```
    forwarding impl only area;
    forwarding impl except circumference;
    ```
  - currently only supported for the standalone declaration form

Forwarding: Impact and Next Steps

Impact:

- Easier to write composition patterns
- Enables support for generic types like 'Owned' and 'Shared'

Next Steps:

- Gain more experience with forwarding
  - Apply to record-wrapper patterns in internal/standard modules
  - e.g., sync/single variables currently manually forward ~12 methods
- Document in the language specification
- Consider improvements to the feature
  - allow 'only' and 'except' in the field declaration form
  - is there a way to forward initializers?
  - can it simplify current instances of iterator forwarding?

First Class Void Variables and Fields

Void Variables: Background

- **The Chapel compiler does not use a preprocessor**
  - No easy way to conditionally declare variables or fields
  - An equivalent to the following C code would be useful in Chapel

```
#ifdef DEBUG
  const char* debugVar = “debug message”;
#endif
…
#ifdef DEBUG
  printf(”%s\n”, debugVar);
#endif
```

Void Variables: This Effort

- **Added 'void' as a first-class type**
- **Variables and fields can have type 'void'**
- **'void' vars can be used in any context expecting 'void'**

  … passing to generic functions that avoid using them inappropriately

  … assigning to other 'void' vars
- **A 'void' var used in a context requiring a value is an error**
  - Such uses can be protected by param conditionals
- **Compiler removes all 'void' vars after reporting any errors**

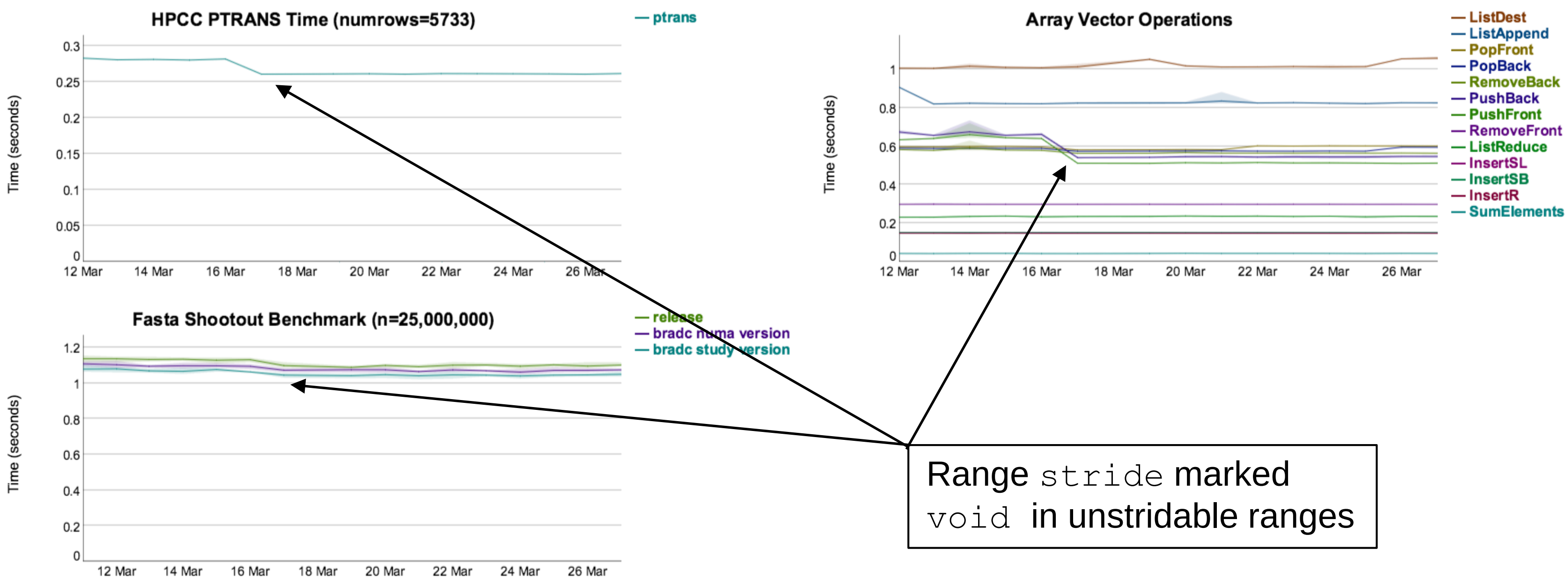
Void Variables: Removing unused fields

- **Declarations can conditionally remove fields**
  - If ‘debug’ is ‘true’, then ’dbgMsg’ is a string available during execution
  - If ‘debug’ is ‘false’ then ‘dbgMsg’ is removed by the compiler

```
param debug = false;
record R {
  var i: int, r: real;
  var dbgMsg: if debug then string else void;
}

var myR = new R(1, 2.3);

if debug then
  myR.dbgMsg = “debugging!”;

writeln(myR);
```

Void Variables: Impact

- **Removing unused fields reduces the size of types**
  - e.g., a range's 'stride' field is not needed if the range is not 'stridable'
    - declaring it 'void' for such cases reduces storage requirements
    - reduced type sizes lead to lower memory footprint and overhead



  - also used to optimize rectangular arrays for non-NUMA locale models

Void Variables: Status and Next Steps

Status:

- ‘void’ is allowed as a first class type in most circumstances
- ‘void’ variables and fields are removed by the compiler
- ranges and rectangular arrays use ‘void’ fields to reduce overhead
- a ’void’ example/primer demonstrating use cases is available

Next Steps:

- Differentiate functions/iterators that don’t return vs. return ‘void’
  - Function with no return is currently treated as returning a single ‘void’ value
- Define and implement more complex types involving ‘void’
  - Arrays with ‘void’ elements
  - Tuples with some/all ‘void’ elements
- Finalize a name for the ‘void’ value
  - Currently using ‘_void’ which doesn’t seem ideal

‘require’ improvements

‘require’: Background and This Effort

Background:

- ‘require’ permits external file dependencies to be expressed in source
  - for example:

```
require "foo.h", "foo.c";
require "bar.h", "-lbar";
```

- traditionally, such requirements…
  …could only be expressed as string literals
  …have been processed whenever they’re encountered in parsed code

This Effort:

- relaxes these constraints:
  - permits requirements to be expressed as ‘param’ string expressions
  - only processes ‘require’ statements in resolved code

‘require’: Impact and Next Steps

Impact:

- requirements can now be expressed more powerfully:

```
config param lib = “foo”,          // permit different libraries to be specified
             debug = false;        // link in debugging library?
require lib+“.h”, lib+“.c”;        // construct names using param expressions
if debug then
  require “-ldebug”;               // only link libdebug in if ‘debug’ is true
…
module MultiThreaded {
  // the following requirements only apply if this sub-module is ‘use’d
  require lib+“_mt.h”, “-l”+lib+“_mt”;
  …
}
```

- used by FFTW and BLAS modules to select between implementations

Next Steps: look for other packages that can benefit from this

Module deinit() functions

Module deinit: Background

- Module initialization is defined by top-level statements
- Global variables are implicitly destroyed at program exit

```
module ModuleExample {
  var globalRecord: MyRecord;
  var globalArray: [1..3] real;
  writeln("done module init!");
  ......
  deallocate globalArray
  globalRecord.deinit()
}
```

run at program start-up

run at program tear-down, implicitly

Module deinit: Background

- **Consider a global class instance:**

```
module ClassExample {
  var globalClass = new MyClass();
  writeln("done module init");
  ......

  delete globalClass
  writeln("deleted globalClass")
}
```

at start-up

want to run this at tear-down

- **How can the user delete 'globalClass' at the end?**
  - recall that deleting class instances is user responsibility
  - Chapel has lacked a convenient way to do that
  - … or to specify other program cleanup actions

Module deinit: This Effort

- **Allow user-defined module deinitialization functions**
  - defined as ‘proc deinit() {...}’ at module level

```
module ClassExample {
  var globalClass = new MyClass();
  writeln("done module init");
  ......
  proc deinit() {
    delete globalClass;
    writeln("deleted globalClass!");
  }
}
```

user-defined module deinit() !

Module deinit: This Effort

- **Clarified + fixed deinitialization order of global variables**
  - happens after user deinit, if present; in reverse declaration order

```
module ModuleExample {
  var globalRecord: MyRecord;
  var globalArray: [1..3] real;
  var globalClass = new MyClass();
  writeln("done module init");
  ......
```

deinitialization order

```
  proc deinit() {
1    delete globalClass;
     writeln("deleted globalClass!");
  }

2  deallocate globalArray
3  globalRecord.deinit()
}
```

Module deinit: This Effort

- **Clarified + fixed deinitialization order of modules**
  - reverse order of module initialization

initialization order

deinitialization order

```
module Main {
  use Helper; ......                                   (init 2)
  proc deinit() { ... }                (deinit 1)
  deinitialize/deallocate Main globals implicitly
}
```

```
module Helper {
  ......                                               (init 1)
  proc deinit() { ... }                (deinit 2)
  deinitialize/deallocate Helper globals implicitly
}
```

  - to see module deinitialization order, compile with
    `-s printModuleDeinitOrder`

Module deinit: Impact and Next Steps

Impact:

- No longer need to wrap module cleanup code in a global record
  - supported simplifications to the MPI module
    - contributed by Nikhil Padmanabhan

Next Steps:

- Apply to other packages
  - e.g. FFTW, FFTW_MT
- Consider adding an optional module init() routine
- Gather user feedback

Improvements to Intents

Improvements to Intents

- ['const' and 'const-ref' as 'this' intents]()
- [Default intent for 'this' on records]()
- [Return intent overload improvements]()
- [Tuple changes]()

'const' and 'const ref' as 'this' intents

'const' and 'const ref' as 'this' intents

Background: *'this' intents* control method receiver arguments

- the method receiver is called 'this' inside the method
- the 'this' intent controls the implicit formal argument for 'this':

```
proc ref int.increment() {
  this += 1; // 'this' is mutable because of 'ref' intent above
}
var x = 1; x.increment();
```

- Chapel has only allowed a subset of intents here:

```
param type ref
```

This Effort: Support 'const' and 'const ref' as 'this' intents

```
proc const ref int.square() { return this*this; }
proc const int.cube() { return this*this*this; }
```

Impact: 'this' intent now supports more cases

Next Steps: Support 'const in' and 'in' as 'this' intents

Default intent for 'this' on records

Default ‘this’ intent: Background

- ***‘this’ intents*** **select a method receiver’s argument intent**
- **Default intent used if ‘this’ intent is not explicitly specified**
  - default intent based on the receiver’s type, as with normal arguments
- **Default ‘this’ intent for records was inconsistent**
  - specified as 'const ref’ but implemented as 'ref'
- **Resulted in several bugs / odd behaviors**
  - methods modifying 'this' could be called on a 'const' record

```
record R {              proc R.reset() {
  var x: int;             this.x = 1;
}                       }
const cR: R;            cR.reset(); // should be an error but was permitted
```

  - method calls on elements of arrays of records used the 'ref' overload
    - led to problems similar to those described in the “array default intent” slides

Default 'this' intent: This Effort

- **Changed the default 'this' intent for records**
  - to 'ref' if 'this' is modified in the method body
  - to 'const ref' if not
- **Rationale:**
  - programmer should be able to omit 'ref' intent as a convenience
  - 'const' should not be required to avoid surprising behaviors
  - compatible with existing Chapel programs
- **For example:**

```
record R {
  var x: int;
}

const cR: R;
var vR: R;
```

```
proc R.reset() {
  // 'this' is modified
  // so 'this' has 'ref' intent
  this.x = 1;
}
vR.reset(); // OK
cR.reset(); // error
```

```
proc R.getX() {
  // 'this' is not modified
  // so 'this' has 'const ref' intent
  return this.x;
}
vR.getX(); // OK
cR.getX(); // OK
```

Default 'this' intent: Impact and Next Steps

Impact:

- Enabled other improvements
  - significant improvements to const-checking
  - fewer surprises with return intent overloads

Next Steps:

- Consider allowing records to select this behavior for all arguments
  - not just implicit 'this' arguments
- and/or: permit records to select between a number of default intents

Return Intent Overload Improvements

Return Intent Overloads: Background

- **Return intent overloads support context-dependent calls:**
  - 'ref' return version is used when call is modified/modifiable
    - typically by passing to a 'ref' formal argument, as with LHS of '='
  - value or 'const ref' return version is used in other cases

```
var x = 1;
proc accessX() ref {    // "setter" version
  return x;
}
proc accessX() {        // "getter" version
  return 0;
}
accessX() = 3;          // uses the "setter" version
writeln(x);             // prints 3
var tmp = accessX();    // uses the "getter" version
writeln(tmp);           // prints 0
```

Return Intent Overloads: Problem

- **We noticed surprising behavior related to locale queries**
- **'ref' version was always selected if the locale was queried:**

```
const ParentDom = {0..10};
var SparseDom: sparse subdomain(ParentDom);
var A: [SparseDom] int;
writeln(A[0].locale.id);
// error: halt reached - attempting to assign a 'zero' value in a sparse array: (0)
```

- **Behavior stems from an implementation workaround**
  - workaround enabled locale queries for arrays of primitive types
  - but, fragile and problematic:
    - didn't handle querying the locale of element passed to fn by 'const ref'
    - inhibited const-checking when querying the locale of a 'const' array element

Return Intent Overloads: This Effort

- **Make return intent overloads handle 'const ref' vs value**
  - 'ref' return version is used as before, when call is modified/modifiable
  - 'const ref' return version is used if passed to 'const ref' formal
    - e.g., when the locale is queried in previous example
  - value return version used otherwise
- **All 3 return intent overloads can now be provided:**

```
var x = 1;
proc accessX() ref {             // "setter" version
  return x;
}
proc accessX () const ref {      // "getter" const reference version
  return x;
}
proc accessX () {                // "getter" value version
  return 0;
}
```

Return Intent Overloads: Status and Next Steps

Status:

- Language change is implemented and specification updated
- Problematic workaround is removed
- Array implementation now uses all 3 return intent overloads
- Motivating example behaves correctly

Next Steps:

- Fix bug with ambiguity in return intent overloads
  - should generate an ambiguity error
  - … but return intent overload currently disabled in this case
- Allow return intent overloads without 'ref' version
  - Currently 'ref' version is required to do return intent overloading
  - 'const ref' and value return overloads are useful on their own

Tuple Changes

Tuples: Background

- **Details of tuple behavior 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
- **Array memory fixes ran afoul of issues with tuples**
- **For example:**

```
proc f( tupleArg ) {
  return tupleArg;
}
var A, B: [1..n] int;
f( (A, B) );
```

  - *are A and B passed by value or by reference into f?*
  - *does returning tupleArg return the contained arrays by value or by ref?*

Tuples: This Effort

- **Reworked the tuple implementation to support array fixes**
  - guiding principle: 1-element tuples behave similarly to plain elements
  - implementation is now more direct and straightforward
- **Updated CHIP 6 to reflect current tuple semantics**

- **Returning to the example:**

```
proc f( tupleArg ) {
  return tupleArg;
}
var A, B: [1..n] int;
f( (A, B) );
```

  - *are A and B passed by value or by reference into f?*
    - by reference, because arrays pass by 'ref' / 'const ref' by default
  - *does returning tupleArg return the contained arrays by value or by ref?*
    - by value, because arrays return by value by default

Tuples: Impact on Program Behavior

- This example behaves differently in 1.14 and 1.15:

```
config const n = 2;
record BigRecord {
  var A: [1..n] int;    // defines a record containing an array field
}
var global: BigRecord;  global.A = 1;
test( (global, global) ); // how does created tuple capture the record?
                          // 1.14: by value, 1.15: by const reference
proc setGlobal() {
  global.A = 9;
}
proc test( tup ) {
  setGlobal();    // does this affect tup(1)?
  writeln(tup);
}
// 1.14 prints     ((A = 1 1), (A = 1 1))
// 1.15 prints     ((A = 9 9), (A = 9 9))
```

Tuples: Next Steps

- Improve const-checking for tuple arguments
- Update language specification with tuple semantics

Improvements for Generics

Generics

- Where-clause improvements
- Type aliases for generic classes
- Secondary methods on instantiated types

Where-clause Improvements

Where-clause: Background

- **"Where-clauses" constrain function candidate choices**

```
proc foo(x) where x.type == int  { writeln("int");  }
proc foo(x) where x.type == real { writeln("real"); }
foo(3); // resolves to the "int" version

proc arrayOp(A: []) where A.rank == 1 { /* optimized 1D case */ }
proc arrayOp(A: [])                   { /* general case */ }
```

- **Useful for…**

  …specializing/constraining functions based on types

  …providing optimized functions

Support where-clauses on non-generic functions

Background: Where-clauses were restricted to generic functions

- didn't see a use-case for non-generic functions originally
  - assumed they would only compute on aspects of function signature (if function is non-generic, there'd be nothing to compute)
- have since recognized value for non-generic functions
  - e.g. selecting function implementation based on a config param

```
config param layout = rowMajor;
proc matrixOp(…) where layout == rowMajor { /* row-major impl */ }
proc matrixOp(…) where layout == colMajor { /* column-major impl */ }
```

This Effort: Support where-clauses on non-generic functions

Status: Generality of where-clauses has been improved

Evaluate where-clauses on matching functions

Background: Historically, where-clauses were always evaluated

- even on functions that didn't have matching signatures
  - could lead to confusing error messages

```
proc foo(param x: int) where (x % 2) == 0 { writeln("even"); }
proc foo(param x: int) where (x % 2) == 1 { writeln("odd");  }
proc foo(param x: real)                   { writeln("real"); }
foo(2.2); // used to generate an error about "%" not being defined on real
```

This Effort: Evaluate where-clauses only for valid arg signatures

Status: Usability and stability of where-clauses has improved

- no known bugs remaining, no future work planned

Type Aliases for Generic Classes

Type Aliases for Generic Classes

Background: Type aliases worked for concrete types only

- a type alias introduces another name for a type
- previously these worked with concrete types, as in:

```
type myint = int;
```

- but they did not work for generic types, as in:

```
type RandomStream = PCGRandomStream;
```

This Effort: Adjust implementation to allow generic type aliases

```
type RandomStream = PCGRandomStream;
```

Next Steps:

- Fix problems with aliases of instantiated types, as in:

```
type RandomIntegerStream = RandomStream(eltType=int);
```

- Adjust Random standard module to use type alias for RandomStream

Secondary Methods on Instantiated Types

Secondary Methods on Instantiated Types

Background: Chapel has *open methods*

- methods can be added to existing types
- but, adding a method to an instantiated generic type required 'where':

```
proc Owned.frobnify() where this.type == Owned(MyClass) { … }
```

- even though the non-method case is straightforward:

```
proc frobnify(arg: Owned(MyClass)) { … }
```

This Effort: Allow parenthesized secondary method declarations

- enables a simpler expression of the above example:

```
proc (Owned(MyClass)).frobnify() { … }
```

- parens required to disambiguate generic class's args from formal args

Impact: Removes a restriction on method receivers vs other args

- where-clause approach no longer required
- simpler syntax is now available

Other Language Improvements

Other Language Improvements

- **Added min() and max() param overloads**
- **Support for casts between c_void_ptr and class objects**
- **Enabled 'param's and 'config param's without initializers**

```
config param x : int;
```

- **First-class functions no longer capture outer variables**
  - Removed as part of separate effort
  - First-class functions support needs redesign/revisiting anyways

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