Language Improvements (Work-in-Progress)

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.

Work-in-Progress Features

- **Initializers**
  - Deinitializers
- **Error-handling**
- **User-Defined Reductions**

Initializers

Initializers: Background

- **Chapel's traditional constructor story was naïve**
  - Became increasingly clear as users/developers relied on OOP more
  - Lacked a good copy constructor / initializer story
  - Implementation generated multiple layers of copy operations

- **Have been developing 'initializers' as a replacement**
  - Consist of two phases:
    - phase 1: constrained initialization of fields
    - phase 2: general computation
  - Phases separated by call to one of:
    - super.init, for initialization of inherited fields (if any)
    - this.init, for common operations on the type (including field initialization)
  - Design being managed in CHIP 10
    - See also the 1.13 release notes on constructors
  - Constructors will be deprecated once initializers are complete

Initializers: Background

- **Last release: initial support for compliant initializers**
  - Supported initializer syntax
    - Included some Phase 1 semantic checks
  - Developed a strategy for generics and copy initializers
  - Further details in the [1.14 release notes]() on initializers

Initializers: Summary of this Effort

- Revised normalization logic for variable declarations
- Implemented initializers as ‘void’ methods
- Improved semantic checks
- Added support for copy initializers
- Started support for initializers on generic classes

Initializers: Background - Variable Declarations

- **Normalization breaks var decls into multiple AST nodes**
  - Resolution processes these nodes independently

  - Comparable declarations generated different AST and C code

```
var r1 : MyRecord = new MyRecord(10, 20);
var r2            = new MyRecord(10, 20);
```

  - But fundamentally both of these were treated as
    1. Default initialize r1/r2
    2. Custom initialize a compiler temp
    3. Copy/Assign the appropriate temp to r1/r2

  - Easy to optimize for primitive scalars but harder for record-like types

Initializers: This Effort - Variable Declarations

- **Refactored the implementation**
  - Includes early type analysis for a few simple cases
    - Initial focus on records with initializers
  - For a record, normalize generates AST to invoke an initializer
    - Function resolution selects the appropriate initializer
  - Simplifies business logic and AST in some cases
    - e.g. the following declarations generate consistent, simplified AST

```
var x : int;
var y = 10;
```

  - Reduces temps/copying for primitive scalars and records

Initializers: Background – Void Methods

- **1.14 implementation leveraged constructor infrastructure**
  - Provided a fast path for initial development, permitting us to…
    …experiment with the syntax
    …develop some initial tests

  - Constructors are effectively functions that return class/record values
    - Contributes to copying overhead since records are returned by copy-out

  - Constructors have special support within function resolution

Initializers: This Effort – Void Methods

- **Revised implementation relies on method infrastructure**
  - Initializers are methods with ‘void’ return type
  - No special support within function resolution for concrete types
  - Normalize gained semantic checks to enforce Phase 1 / Phase 2 rules
  - Methods receive a ‘ref’ argument to the object
    - No copying

Initializers: This Effort – Other Improvements

- **Errors are reported when**
  - .init calls or field initializations are present in loops
  - .init calls or field initializations are present in parallel code
  - any field initialization occurs prior to this.init()
  - a field is accessed before it is initialized in Phase 1
    - some exceptions to this persist
  - a const field is assigned in Phase 2

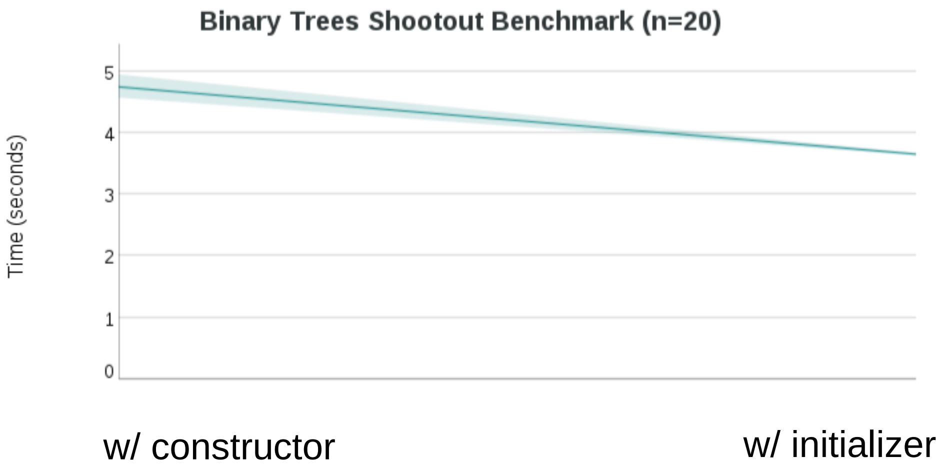
- **Conditional statements in Phase 1 now supported**
  - Initialization must be consistent across branches
  - If either branch includes a .init() call then both must do so
  - Compiler ensures omitted fields are initialized consistently

Initializers: This Effort – Generics

- **Support initializers for classes with generic fields**
  - type fields, param fields, vars/consts with neither types nor initializers
- **Appropriate initializer selected by function resolution**
- **Generic fields handled consistently with non-generics**
  - If initialization of a generic field is omitted in Phase 1…
    …if provided, the default field initializer is used
    …otherwise, an error is generated
  - Cannot update param or type fields in Phase 2
- **Instantiated / concrete type is known by end of Phase 1**
  - Creates new concrete type if necessary
    - Instantiates methods for the instantiated type

Initializers: Impact

- **Converted the Binary Trees shootout to use initializers**
  - Resulted in ~1.27x performance improvement
  - Resulted in cleaner generated code
    - Constructor code was 59 lines, 2 functions, set fields 3 times
      - Couldn’t write with explicit constructor due to a recursion bug
      - Relied on default constructor and special factory type method
    - Initializer code is 23 lines, 1 function, sets fields 2 times
      - Has a Phase 2 body. If implemented using Phase 1, would only set the fields once



Initializers: Status

- **Implementation has improved**
  - At the release of Chapel 1.14.0, had 46 passing tests and 25 futures
    - Passing tests tended to cover more basic features
    - Most futures captured important missing features
  - Today, have 131 passing tests and 40 futures
    - Passing tests growing in complexity
    - 31 of the futures are new (77.5%)

- **Simple tests work well**
  - Including most verification of initializer rules, simple generic classes
  - Some edge cases to fix

Initializers: Known Limitations

- **No support for conditional expressions:**

```
class Foo {
  const x: real;
  proc init(flag: bool) {
    x = if flag then 1.2 else 3.4;
  }
}
```

  - this case is now fixed on master

- **Fail to handle field initializers that include params**
  - Any initializer that omits initialization of the field 'v' will fail

```
class Foo {
  param p = 11;
  var   v = p + 4;
  …
}
```

Initializers: Known Limitations

- **Limited support for conditional statements**
  - Compiler may fail to apply phase 1 rules correctly along both branches
  - The following initializer should be accepted
    - A field is initialized before super.init() on then-branch
    - No field is initialized before this.init() on else-branch
    - Erroneous error claims that a field is initialized before a use of this.init()

```
class Foo {
  …
  proc init(a: bool) {
    if a then {
      x = 11; // proper field initialization prior to super.init() call
      super.init();
    } else {
      this.init(); // but this.init() in other branch causes issues
    }
  }
}
```

Initializers: Known Limitations

- **Copy-initializers vs. generic initializers**

```
record Foo {
  …
  proc init(arg) { // a generic initializer
    x = arg.someMethod(); // requires that arg implements someMethod()
  }
}
```

- Compiler may need the copy-initializer for Foo
- Compiler will instantiate this generic initializer as the copy-initializer
- Confusing error message if Foo does not implement **someMethod**()
  - May not be clear where compiler needs the copy-initializer

Initializers: Known Limitations

- **Incomplete support for generic types**
  - Compiler relies on body of the resolved initializer to construct type

```
var r1 = new MyGenericClass(10, 20.0);
```

  - Compiler fails to construct type when there is no initial value e.g.

```
var r2 : MyGenericClass(int, real);
```

- **Generic records not yet well-supported**

- **Other known limitations:**
  - Secondary initializers might be ignored in some cases
  - Poor error message when using field in argument list
  - No support for initializers for nested class/record
  - …(see futures)…

Initializers: Next Steps

- **For initializers on non-generic types:**
  - Implement missing features
    - Address current futures
    - Develop more tests
  - Improve validation for Phase 1 and Phase 2
  - Some general streamlining of AST within any method

- **For initializers on generic types:**
  - Support generic class variables with type specifications, e.g.:

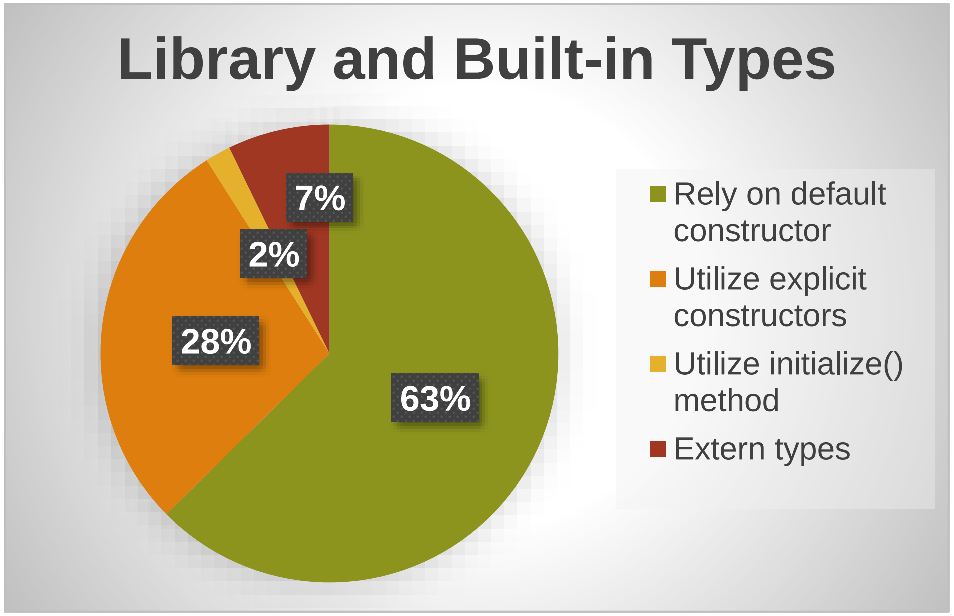
    ```
    var r1 : MyGenericClass(int, real);
    ```

  - Improve support for generic records

- **Determine whether objects should support ‘ref’ fields**
  - If so, add initializer support for them

Initializers: Next Steps

- **Update library and built-in types to use initializers**
  - See chart for break-down of the 222 library and built-in types
  - ~12 types with default constructors have special compiler support
    - will look into retiring that support
- **Generate default initializers and not default constructors**
- **Add support for noinit**
- **Convert constructor tests**
- **Deprecate constructors**



Initializers: Next Steps

- **Re-evaluate design decisions as we convert existing code**
  - Multiple syntax choices
    - e.g., distinguishing phases 1 and 2
  - What should the default phase be?
  - Could the distinction between phases be blurred in common cases?
    - e.g., support params / types in phase 2 since evaluated at compile-time?
  - Should ‘const’ fields be re-assignable in phase 2?
  - Generation of default initializers (currently squashed by user’s init())
    - Should user be able to opt-into retaining compiler’s default init()?
    - Does an initializer for a generic type constrain the possible types?
  - Importance of possible optimizations

Deinitializers

Deinitialization: Background

- **Deinitialization is the opposite of initialization**
  - actions to take when done with resources
    - e.g., release memory, close files
  - taken upon deleting a class, a record leaving its scope, etc.
    - implicitly includes deinitialization of class/record fields, array elements

- **Previously defined using '~typename' methods**

```
record MyRecord {
  ...
  proc ~MyRecord() { ... }
}
```

old-style naming

Deinitialization: This Effort

- **New name for deinitializers: ‘deinit’**

```
record MyRecord {
  ...
  proc deinit() { ... }
}
```

consistent with new terminology and initializer naming

  - if not specified by user, deinit() with empty body is added implicitly

- **Deinitialization and object fields:**

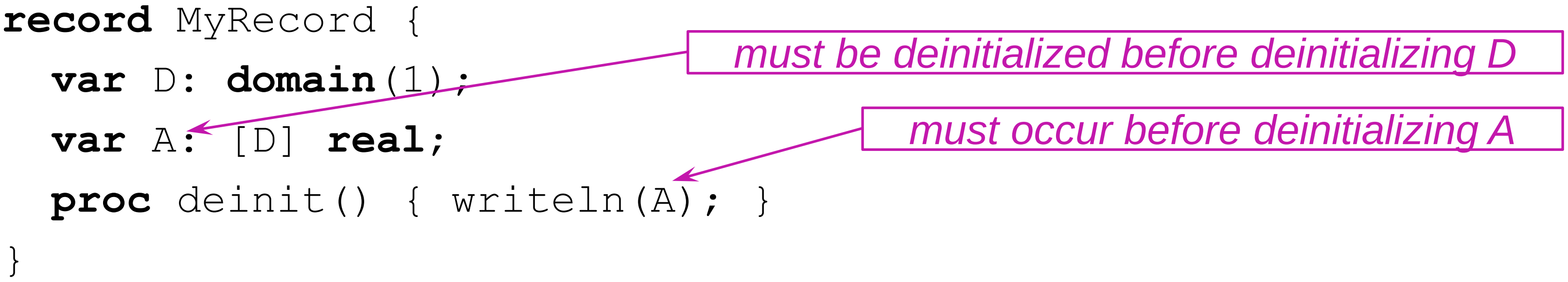
the main expected use case of deinit()

  - field type is a class ⇒ user must ‘delete’ it as appropriate
    - compiler does not add implicit ‘delete’
  - field type is a record ⇒ user cannot deinitialize it
    - deinitialized implicitly by compiler after executing deinit()

- **Introduced initial version of memory safety rules**

Deinitialization: Safety Rules – Overview

Motivation: prevent access to already-deinitialized fields



```
record MyRecord {
  var D: domain(1);
  var A: [D] real;
  proc deinit() { writeln(A); }
}
```

must be deinitialized before deinitializing D

must occur before deinitializing A

- akin to preventing access to not-yet-initialized fields at initialization
- also reduces risk of access to already-’delete’d class fields

Initial version: provides one way to achieve safety

- seeking user feedback
- initial rules are more restrictive
  - gives us room to relax if desired

Deinitialization: Safety Rules – Deinit Order

order of deinitialization actions:

```
  class Parent {
④ var field1...;
③ var field2...;
② proc deinit() {...}
  }
```

```
  class Child : Parent {
    var field3...;
① proc deinit() {...}
  }
```

before or after
2, 3, 4

- run deinit() in child class before deinit() in parent
- fields deinitialized after the deinit() for their class/record
  - in reverse declaration order
  - no user-visible effects for fields of primitive or class types

Deinitialization: Safety Rules – Restrictions

restrictions on deinit() methods:

- cannot invoke methods on ‘this’
- cannot pass ‘this’ as an argument to a procedure
- *can* access individual fields
  - including fields in superclass(es), if applicable

Deinitialization: Status and Next Steps

Status:

- both old and new naming can be used with 1.15
- ‘deinit’ name is used uniformly by Chapel code in repo
- memory safety rules: initial version is developed, partially implemented

Next Steps:

- refine the safety rules as necessary, complete their implementation
- improve error checking:
  - calling methods on ‘this’
  - passing ‘this’ to a function
- deprecate ‘~typename’ along with constructors

Error Handling

Error Handling: Background

- **Chapel had no language-level strategy for handling errors**
  - 'halt()' and "error" out arguments are used in practice, but insufficient
- **Language support was designed but unimplemented**
  - CHIP #8 proposed 'try', 'throws', 'throw', etc.
  - Design used Swift's error handling as a starting point

Error Handling: This Effort

- **Create a draft implementation of the design**
  - Draft offers basic functionality
  - Not yet integrated with tasks, 'on' statements
- **Seek feedback on design and implementation**
  - Encourage users to try the new error handling features

Error Handling: errors as classes

- **Base class 'Error' is provided**
  - For now, the initializer accepts a string argument

```
class Error {
  var msg: string;
}
```

- **'Error' may be used directly, or as the root of a hierarchy**
  - Standard set of 'Error' subclasses not currently included

```
class MyError: Error {}

class MyIntError: Error {
  var i: int;
}
```

Error Handling: throwing errors

- Throw an error with 'throw'

```
// throwing a newly created error
throw new Error("error message here");

// throwing an error stored in a variable
var e = new Error("test error");
throw e;
```

- Mark procedures that can throw with 'throws'

```
proc mayThrowErrors() throws { ... }

proc mayThrowErrorsAlso(): A throws where { ... }

proc mayNotThrowErrors() { ... }
```

Error Handling: try/catch

- **'try' and 'try!' are used to handle thrown errors**
  - { } blocks try to match to an associated 'catch' clause
  - Single statements will not match any 'catch' clauses

```
try {
  mayThrowErrors();
  mayNotThrowErrors();      // non-throwing calls may be included
  mayThrowErrorsAlso();
}
try! mayThrowErrors();      // halts on error
```

- **If an error is handled with no matching 'catch' clause:**
  - 'try' propagates the error
    - To an outer 'try', or out of the procedure (which must be marked 'throws')
  - 'try!' halts instead of propagating
  - Single statement form relies on this behavior

Error Handling: try/catch

- **'catch' clause list matches against an 'Error' at run-time**
  - If a type filter matches the error, that block will be executed
  - Lack of a type filter means that all errors match

```
try {
  trickyOperation(badArg);
} catch err: IllegalArgumentError {  //  IllegalArgumentError, subtypes
  writeln("illegal argument!");
} catch err: MyError {               //  MyError, subtypes
  throw err;
} catch {                            // catch-all
  writeln("unknown error!");
}
```

Error Handling: default and strict mode

- **Two modes to support the tradeoff between…**

  …ensuring propagation of errors is clear (strict)
  …drafting code quickly (default)

- **Strict mode enforces visible control flow**
  - All calls to throwing procedures must be enclosed within 'try' / 'try!'
  - Otherwise, an error will be raised at compile-time

- **Default mode supports rapid prototyping**
  - Throwing calls need not be enclosed in 'try' / 'try!'
  - If the enclosing procedure is marked 'throws', propagate errors
  - Otherwise, halt on errors

- **Strict mode enabled with a compiler flag, --strict-errors**
  - Otherwise, compiler uses default mode
  - Expect to support more fine-grained approaches in the future
    - e.g., specify strictness per-module (or even per-function?)

Error Handling: limitations

- **Cannot span ‘begin’ / ‘cobegin’ / ‘coforall’ / ‘forall’ / ‘on’…**
  - No problems if error handling is kept entirely within the construct

```
begin {
  try {
    mayThrowErrors();
  } catch {
    writeln("handled internally");
  }
}
```

- **Halting errors do not yet print their type or message**
- **Virtual methods cannot yet throw**
- **Errors cannot yet be generic classes**

Error Handling: Status and Next Steps

Status:

- Basic implementation is in Chapel 1.15
  - Soliciting community feedback

Next Steps:

- Address the limitations on the previous slide
- Create a standard set of 'Error' classes
- Enable throwing errors from iterators
- Implement the 'defer' construct for state cleanup
- Design and implement a fine-grained strict mode
- Integrate error handling into the standard library
- Handle runtime errors by throwing Chapel errors

User-Defined Reduction Interface

Reduction Interface: Background

- **Chapel allows custom reduction operations…**

```
var myVar: real;
forall a in A with (MyReduceOp reduce myVar) do
  myVar reduce= a;
writeln(myVar);
```

… to be implemented using *reduction classes*

```
// today, a custom plus-reduction class might look like this
class MyReduceOp: ReduceScanOp {
  type eltType;                        // type of input
  var  value: eltType;                 // accumulation state
  proc identity return 0:eltType;
  proc accumulate(input) { value += input; }
  proc combine(childOp) { this.value += childOp.value; }
  ...
}
```

Reduction Interface: Background

- **Reduction class implements details of reduction**
  - for example:
    - identity value
    - how to *accumulate* an input value into the accumulation state
    - how to *combine* two accumulation states
  - they are invoked by compiler for reduce expressions and intents
  - the same interface is used for standard reductions (+, max, etc.)
- **Compiler works with reduction class to prevent data races**
  - compiler ensures no race when *accumulating* the input values
  - *combining* needs to lock the parent task's accumulation state

```
const parentOp = new MyReduceOp(...);
coforall task in 1..numTasks {
  const childOp = new MyReduceOp(...);
  ... accumulate input onto childOp ...
  parentOp.combine(childOp);
}
```

multiple child tasks may try to combine onto the same parent task at once

Reduction Interface: New Requirements

- **Separate accumulation state for reduce intents**

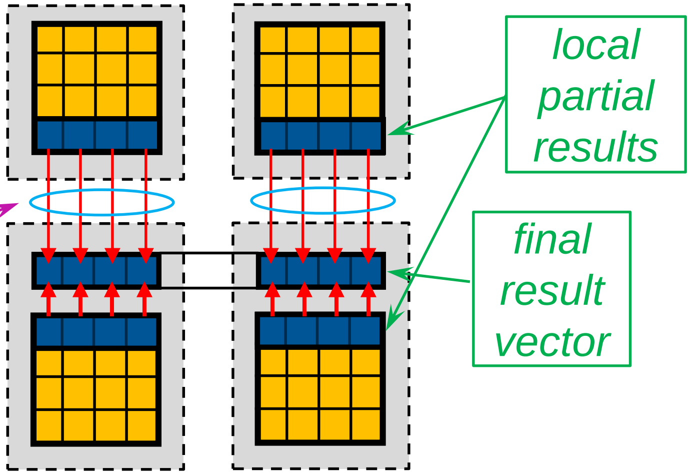
```
forall ... with (MyReduceOp reduce myVar) {
  myVar reduce= some input;
}
```

accumulation state to be accessible via shadow variable

- **Avoid lock state for partial reductions (w.i.p.)**

example: partial reduction of a Block-distributed matrix to a vector

for bulk communication of local results, want to have accumulation state without locks



- **Support synchronization strategies other than locking**
  - e.g., atomics

Reduction Interface: This Effort

Revisit the reduction interface in light of these requirements

- **Reduction class to be stateless**
  - instead, will provide methods to create accumulation state
- **Flexible synchronization strategy**
  - When accumulation state is created with argument parSafe=true
    - reduction class chooses/implements synchronization strategy
    - accumulation state includes lock/atomic/...
  - Otherwise, when parSafe=false
    - no need to include locks/atomics in accumulation state
    - this mode will be used for partial reductions

Reduction Interface: Impact

New interface definition

```
class MyReduceOp {         // no required parent class
  type inputType;          // distinct from the type of accumulation state

  // create accumulation state at top level
  // for reduce intents, seed it with value of user variable, otherwise with identity
  proc newGlobalAccState(initUserValue, param parSafe)  ...
  // create accumulation state in child task, initialized to identity
  proc newLocalAccState(ref parentState, param parSafe)   ...

  // accumulate and combine operations
  proc accumulate(ref parentState, ref localState, input) ...
  proc combine(ref parentState, ref localState)           ...

  ...
}
```

Reduction Interface: Impact

Example of compiler-generated calls to the interface

```
// a child task while performing a reduction
proc childTask(reduceClass, ref parentAS)
{
  // executed upon task startup
  var childAS = reduceClass.newLocalAccState(parentAS,
                                             parSafe=true);
  // generated for Chapel statement: userVar reduce= input;
  reduceClass.accumulate(parentAS, childAS, input);

  // if the child task has nested task constructs
  grandchildTask(reduceClass, childAS);

  // executed upon task tear-down
  reduceClass.combine(parentAS, childAS);
}
```

the reduction class instance

parent task's accumulation state

accumulation state to include lock/atomic/...

Reduction Interface: Status and Next Steps

Status:

- New interface design draft is available in issue #5470
- Compiler and Chapel code are still using the previous interface

Next Steps:

- Finalize the new interface
  - check that it works with partial reductions (w.i.p.)
- Implement the new interface
  - adjust the code base
  - fine-tune the interface based on experience
- Reduction record instead of reduction class?
  - well-scoped lifetime ⇒ avoid malloc/free

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