Language Improvements

Chapel Team, Cray Inc.

Chapel version 1.9 summary

April 17th, 2014 (released) / May 2014 (documented)

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

- Updated Assignment Signature
- ‘noinit’: Squashing default initialization
- ‘ref’ intents for ‘this’
- Dynamic casts of ‘nil’
- Break out of ‘param’ for-loops
- Fixing Domain Literals of Domain Values
- Expression-free Serial Statements

Updated Assignment Signature

Updated Assignment Signature

Old-style assignment:

- Took two arguments, but returned the RHS:

```
proc =(a: int(?w), b: int(w)) return b;
```

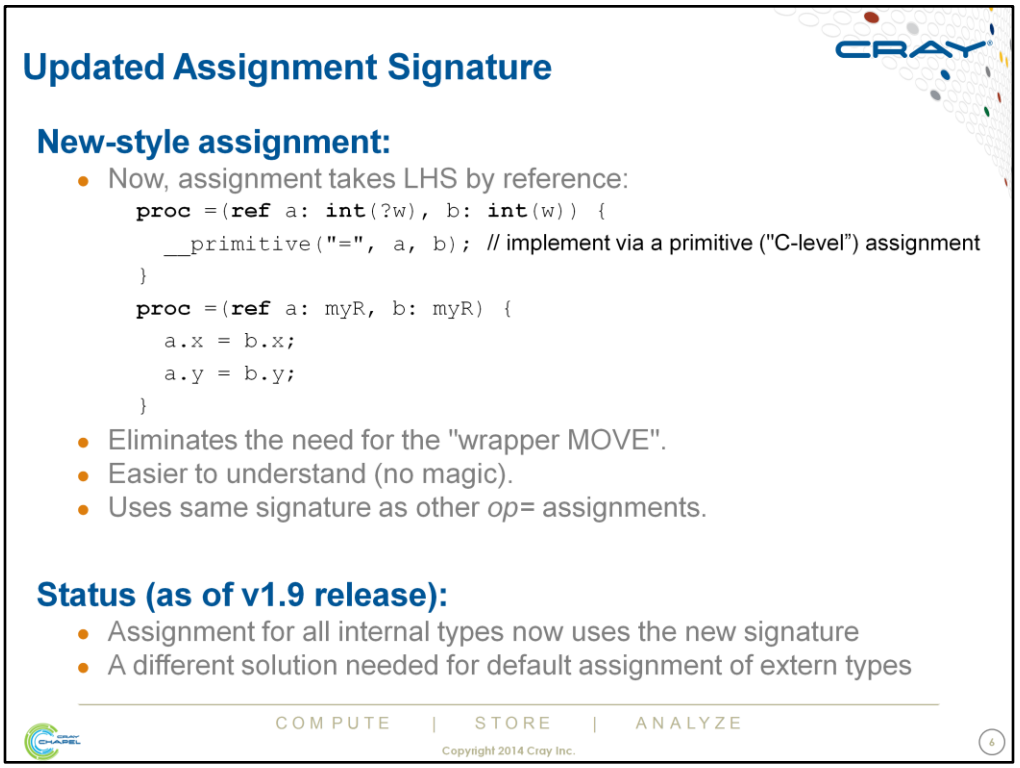
- Rationale:
  - Historically, Chapel did not have 'ref' intents
  - Therefore, couldn't modify 'a' without incurring copies via '[in]out' intent
- Compiler automatically wrapped calls to assignment in a MOVE
  - Necessary to copy the result back into the LHS arg:

```
lhs = rhs;
```

    was translated into:

```
('move' lhs ("=" lhs rhs))
```

- Problems:
  - Requires creating a temporary result of the same type as the lhs.
  - Then requires internal tricks to avoid a verbatim copy-back.
  - "Cognitive dissonance" with other languages
    - also with '*op*=' assignments which had previously switched to using 'ref' intents
  - Larger internal representation



- The __primitive("=", …) call is an implementation-level hook for getting access to "primitive" (below Chapel-level) capabilities. In this case it corresponds to a C-level assignment or memcpy() or potentially a remote put/get. In subsequent slides, it's referred to as PRIM_ASSIGN (the compiler-internal name for it).

- The "MOVE" referenced on this slide (and the previous) is another Chapel primitive (PRIM_MOVE). It plays a similar role as PRIM_ASSIGN, but is considered a bit bloated / dated at present, so the approach being taken with this work is to introduce PRIM_ASSIGN, do it right/well, and then slowly phase out all references to PRIM_MOVE.

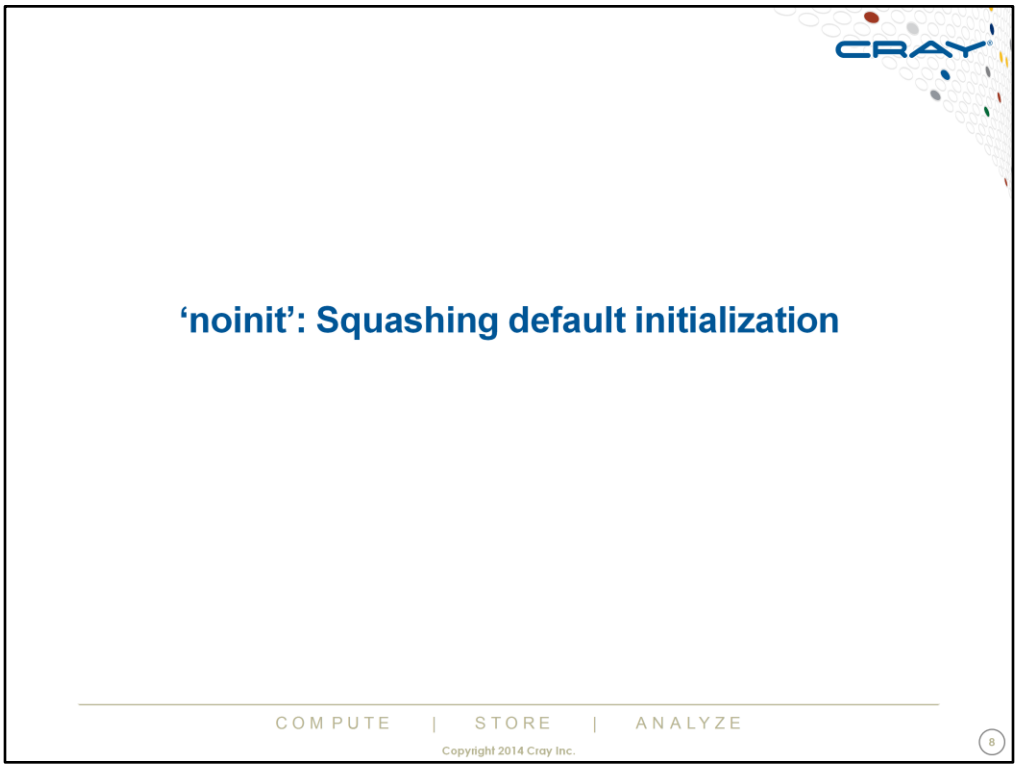
Updated Assignment Signature

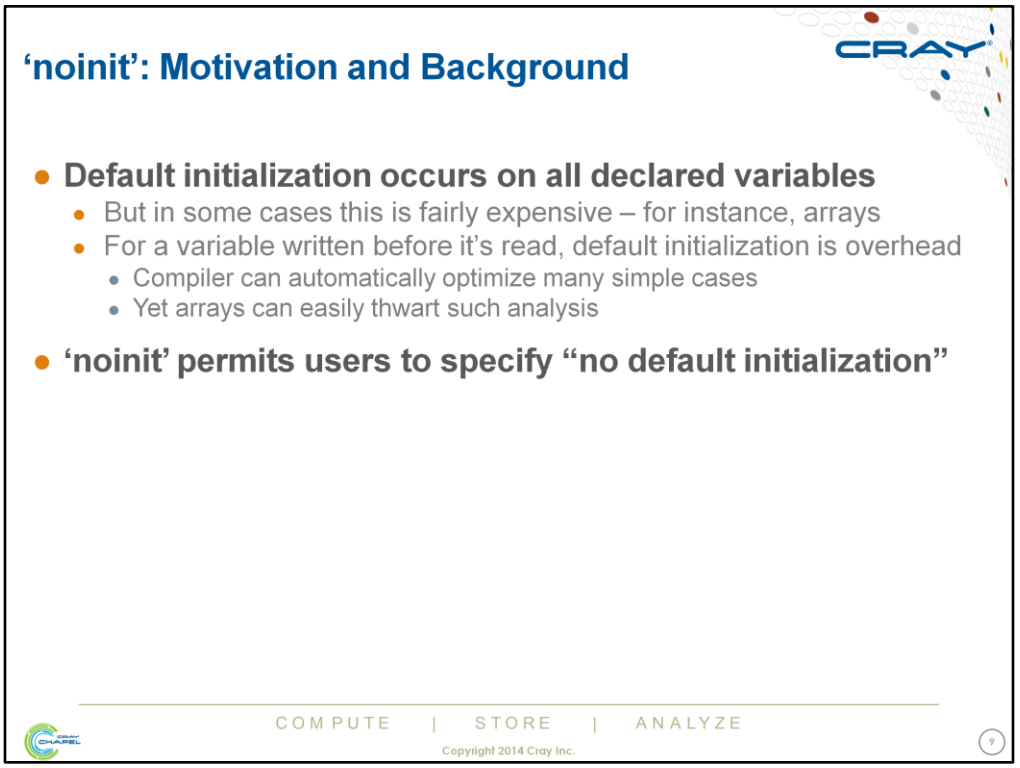
Additional Work Completed Since 1.9 Release:

- Provided default assignment for extern types
- Removed catch-all assignment definition
- Removed compiler insertion of wrapper MOVEs
- Upgraded PRIM_ASSIGN to handle operands of wide class types

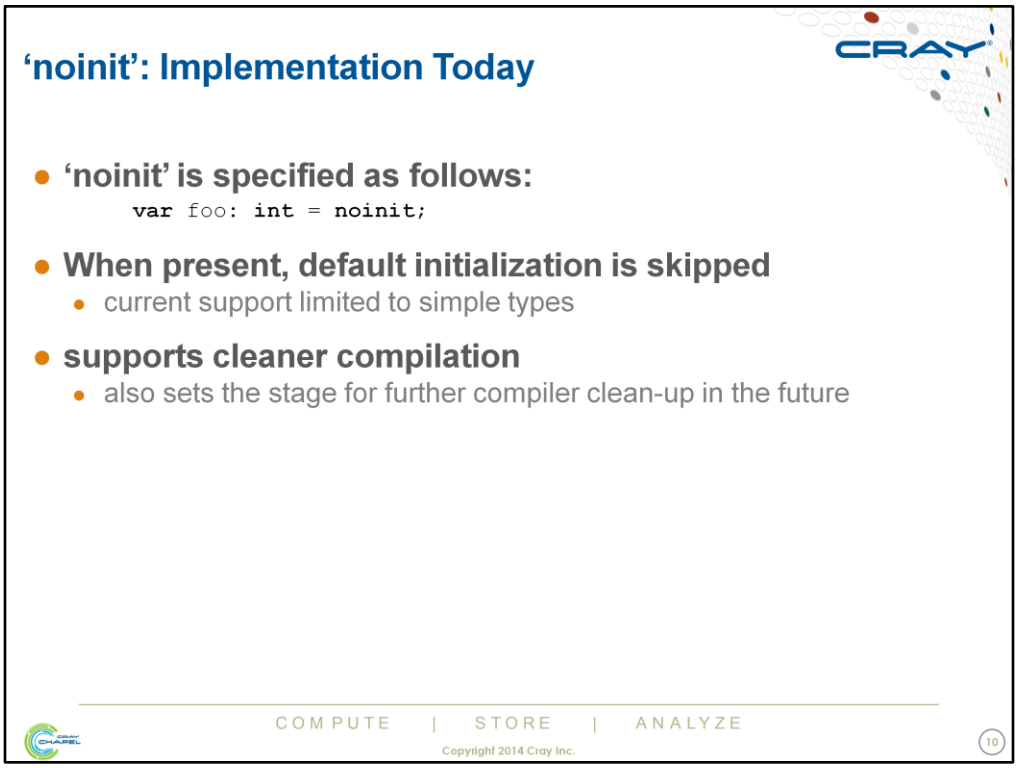
Next Steps:

- Port data motion optimizations to PRIM_ASSIGN
- Eliminate primitive MOVE entirely

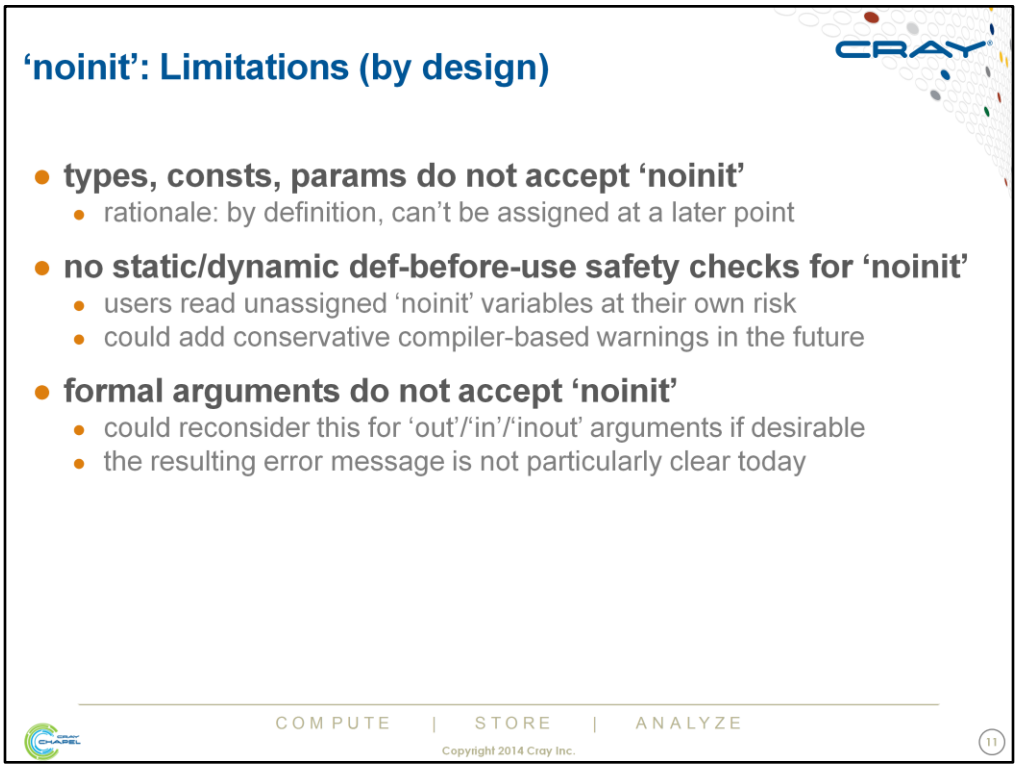

‘noinit’: Squashing default initialization
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
8



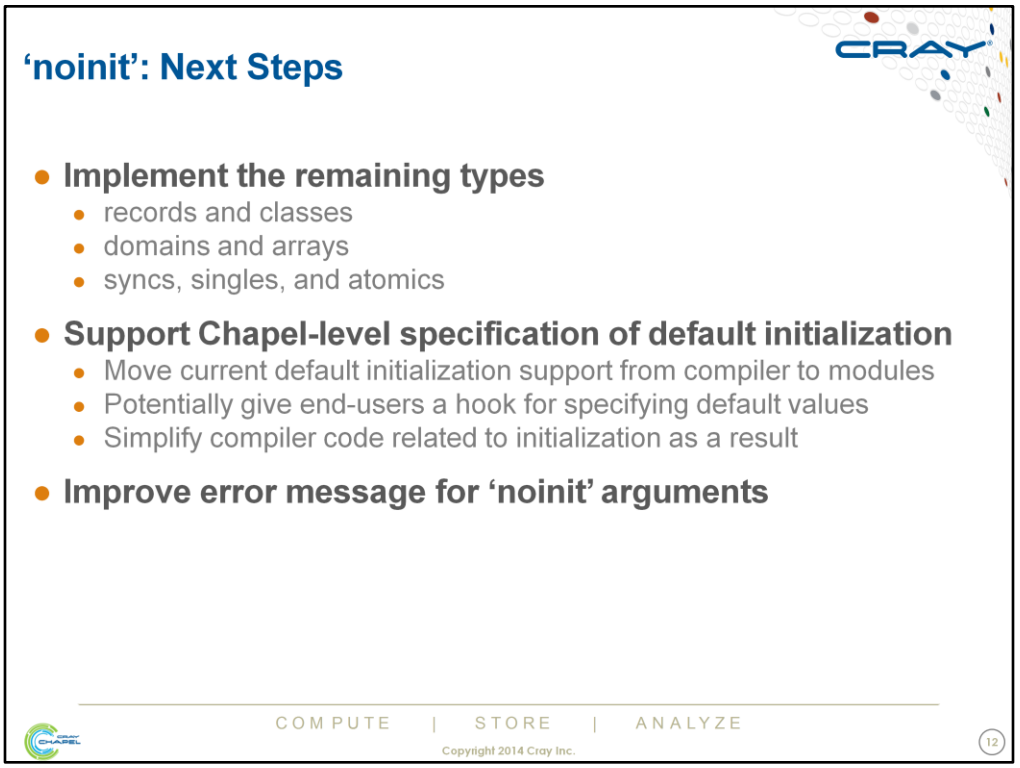
- Arrays present a challenge due to indexing. For example, given "A[0..n/2] = 1; A[m..n] = 2;", is A initialized safely enough for the compiler to squash the default initialization?
- Limits of interprocedural analysis/alias analysis are other limiting factors



- At present, this only occurs for primitive types such as integers and strings; enums; tuples; and ranges
- Certain temporary variables necessary for default initialization are not inserted into the compiler when this keyword is specified, requiring less clean up in simple cases



There is also no dynamic user query to see whether a 'noinit' variable has been initialized or not



- Generally speaking, for the remaining types, the work involves separating the mechanisms used to allocate space from those used to initialize it – currently, these are entangled.
- This work also sets us up for better "first-touch" and parallel initialization policies for arrays in a way that will better match subsequent iterations than the current very naïve scheme.

‘ref’ intents for ‘this’

Allow ‘this’ to be marked with a ‘ref’ intent

Background: Allow modifications of ‘this’ for value types

- Previously, this was possible for developers via ‘pragma “ref this”’

```
pragma "ref this"
proc int.triple() {
        this *= 3;
}
var x = 2;
x.triple();      // x == 6 after this call
```

- We also had language support for marking ‘this’ with a **param** intent

```
proc param string.length param {
        ...
}
"hello there!".length    // results in 12 at compile time
```

Allow ‘this’ to be marked with a ‘ref’ intent

This Effort: Bring ‘pragma “ref this”’ into the language

- We now reuse the syntax for ‘**param** this’ to allow for **ref** or **param**

```
proc ref int.triple() {
        this *= 3;
}
var x = 2;
x.triple();      // x == 6 after this call
```

Impact: Reduced complexity within the compiler

- Removed another pragma and FLAG_REF_THIS
- The implicit ‘this’ argument is now marked with the proper intent
  - This was the source of several special cases

Allow ‘this’ to be marked with a ‘ref’ intent

Next Steps:

- Make this feature illegal for reference types
  - Currently there are no restrictions for marking ‘this’ as ref

```
class Foo { ... }
proc ref Foo.change() {
        this = new Foo();    // ??
}
```

  - In the future this should be an error
- Consider adding other intent types to ‘this’ as well
  - Including more formally defining what a ‘blank’ this intent implies?

Dynamic casts of ‘nil’

Dynamic casts of 'nil'

Background: Dynamic casts of 'nil' caused a segfault

```
class Node { ... }
class SpecialNode : Node { ... }
proc walk(tree: Node) {
        walk(tree.left);
        walk(tree.right);
        var x = tree: SpecialNode;      // segfaults if tree == nil
        ...
```

- This aspect of the language was never defined

Improvement: Dynamic casts of 'nil' always produce 'nil'

- Could be considered a cast failure, which also returns 'nil'

```
...
var x = tree: SpecialNode;      // x == nil or x == valid SpecialNode
...
```

Break out of ‘param’ for-loops

Break out of 'param' for-loops

Background:

- The break statement can be used to exit a loop early

```
for i in 1..10 do
  if shouldExitEarly(i) then break; else f(i);
```

- 'param' for-loops are fully unrolled so each index is a 'param' value

```
for param i in 1..4 do
  f(i);                    // this loop is equivalent to: f(1); f(2); f(3); f(4);
```

- But break has not been supported in 'param' for-loops

This Effort:

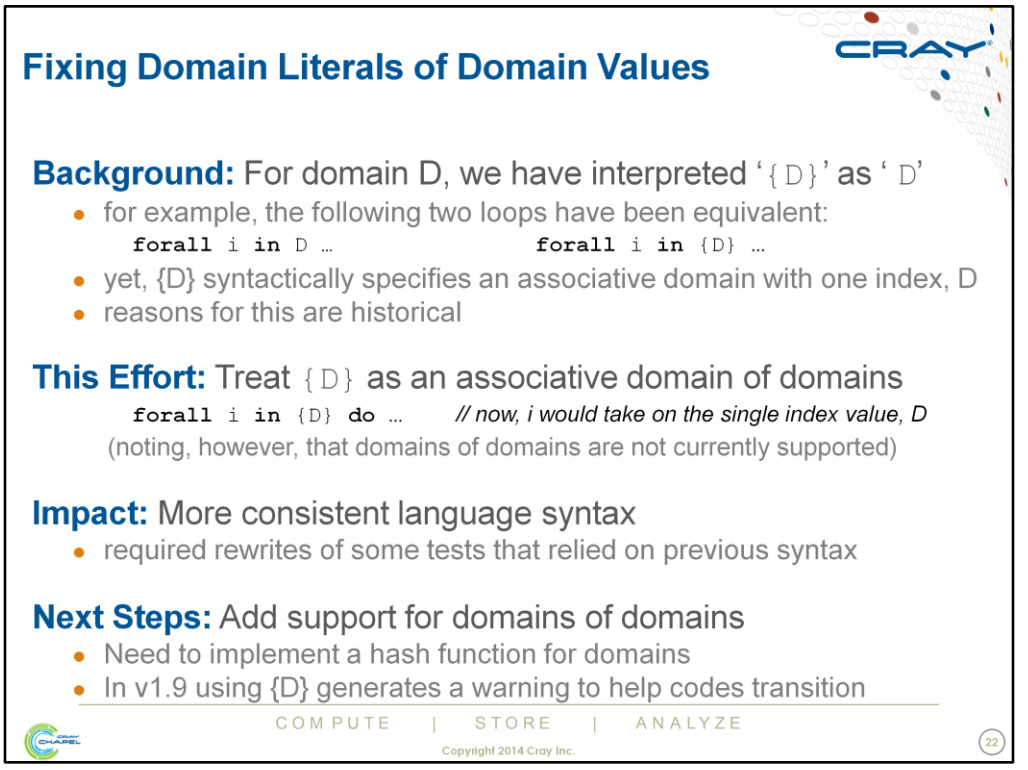
- Support break to allow param for loops to exit early:

```
for param i in 1..10 do
  if shouldExitEarly(i) then break; else f(i);
```

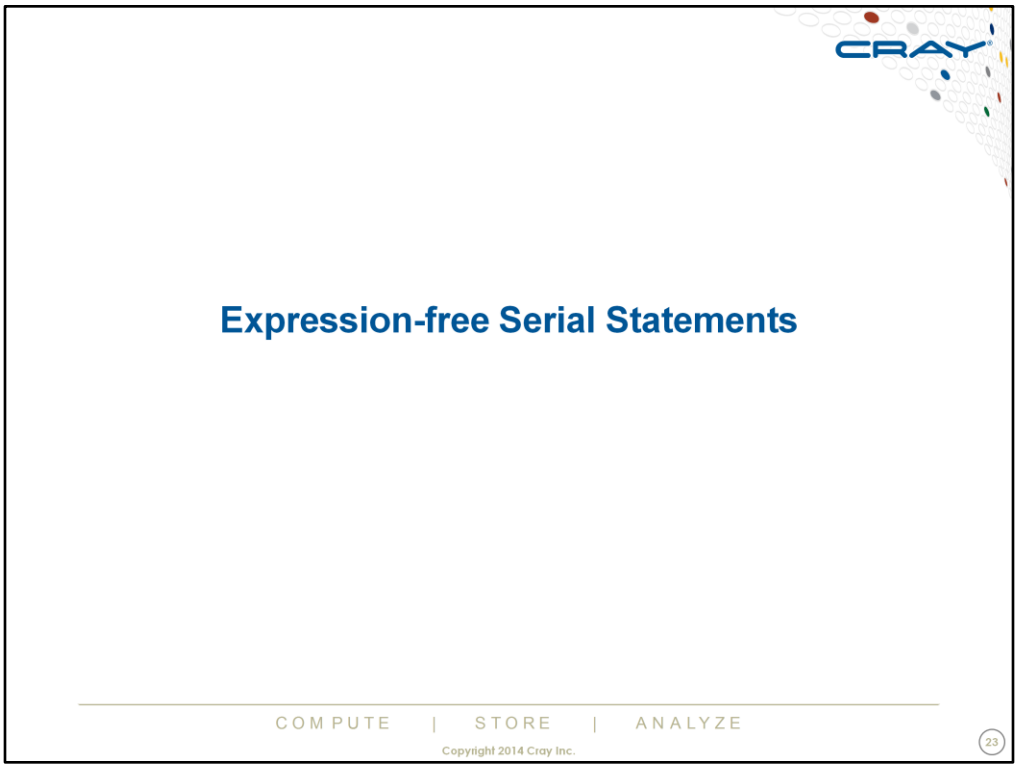
Impact:

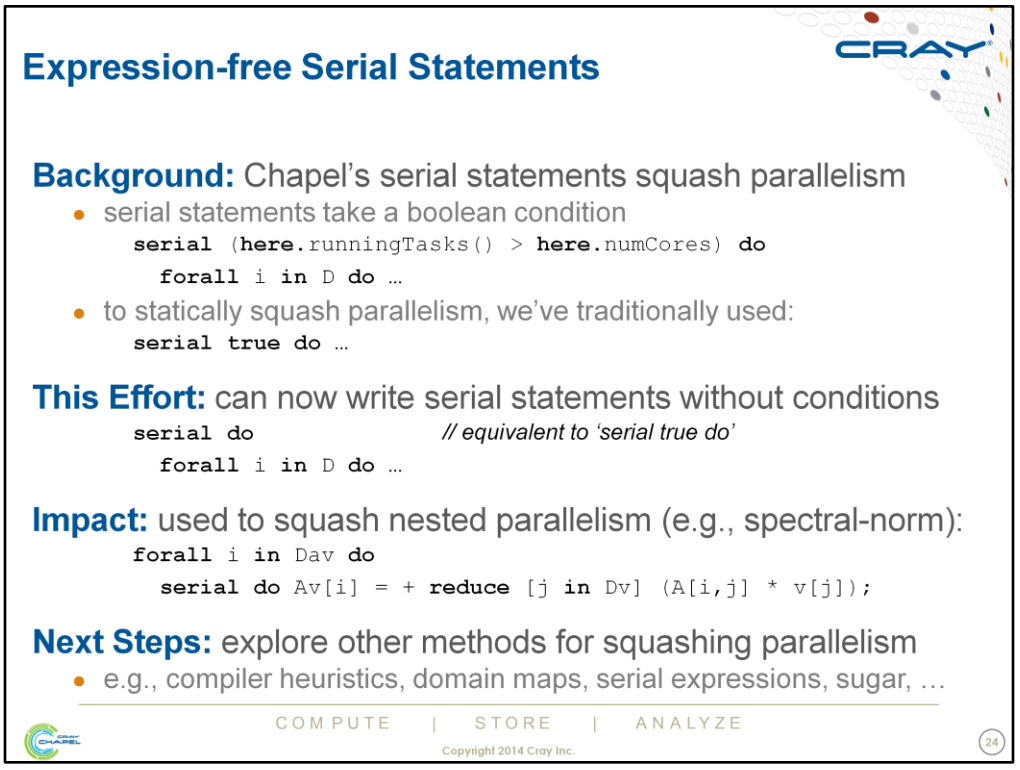
- 'param' for-loops now treated more uniformly with other loops

Fixing Domain Literals of Domain Values



- Prior to this work, both the loops 'forall i in D' and 'forall i in {D}' would have caused 'i' to take on the indices of 'D'.
- This change sets us up to treat the former the same, and the latter as an iteration over an associative domain of domains.
- The historical reasons for this relate to the fact that we used to use square brackets for domain literals, combined with some holdover from ZPL design decisions
- We're not aware of any deep reason that associative domains of domains are not currently supported – simply that nobody has gotten to them / needed them yet. Writing a hash function for domains is the first step (and possibly the only one).


Expression-free Serial Statements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
23



It's worth noting that while some Chapel constructs, like forall loops, are fairly easily to statically serialize (by changing to a for loop, for example), cases like reductions and promoted operations across arrays are more cumbersome to do (requiring converting either case to an explicit loop rather than relying on the more direct expression), and so in these cases 'serial true' tends to be more useful.

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.


CRAY
CHAPEL
CRAY
THE SUPERCOMPUTER COMPANY
http://chapel.cray.com
chapel_info@cray.com
http://sourceforge.net/projects/chapel/