Performance / Memory Improvements

Chapel Team, Cray Inc.

Chapel version 1.9

April 17th, 2014 (released) / May 16, 2014 (documented)

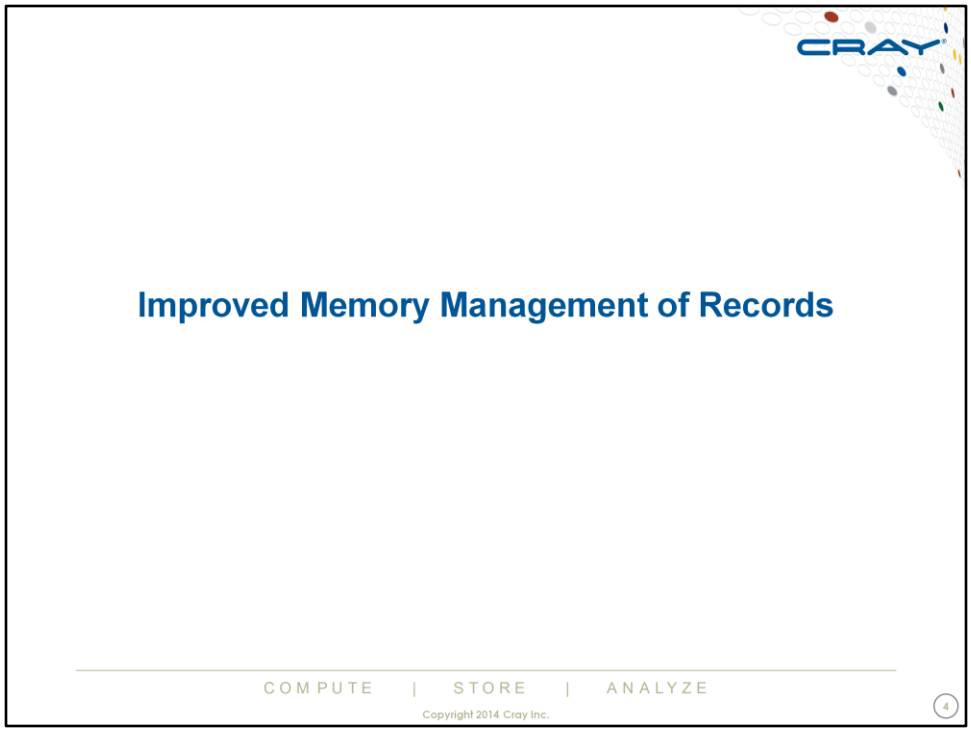
COMPUTE | STORE | ANALYZE

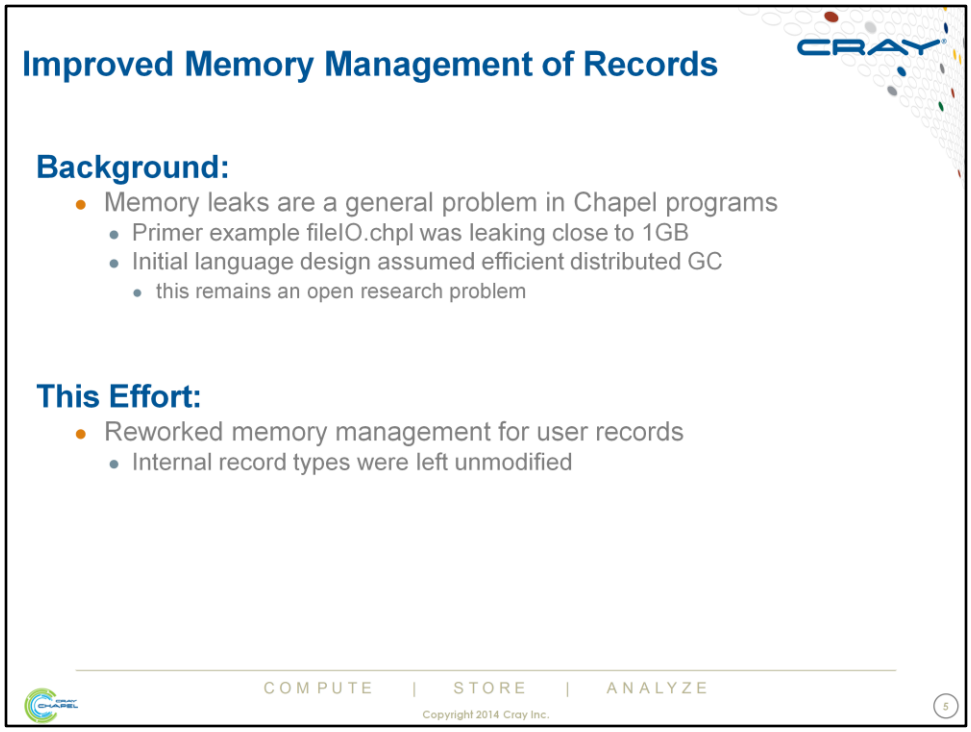
Safe Harbor Statement

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.

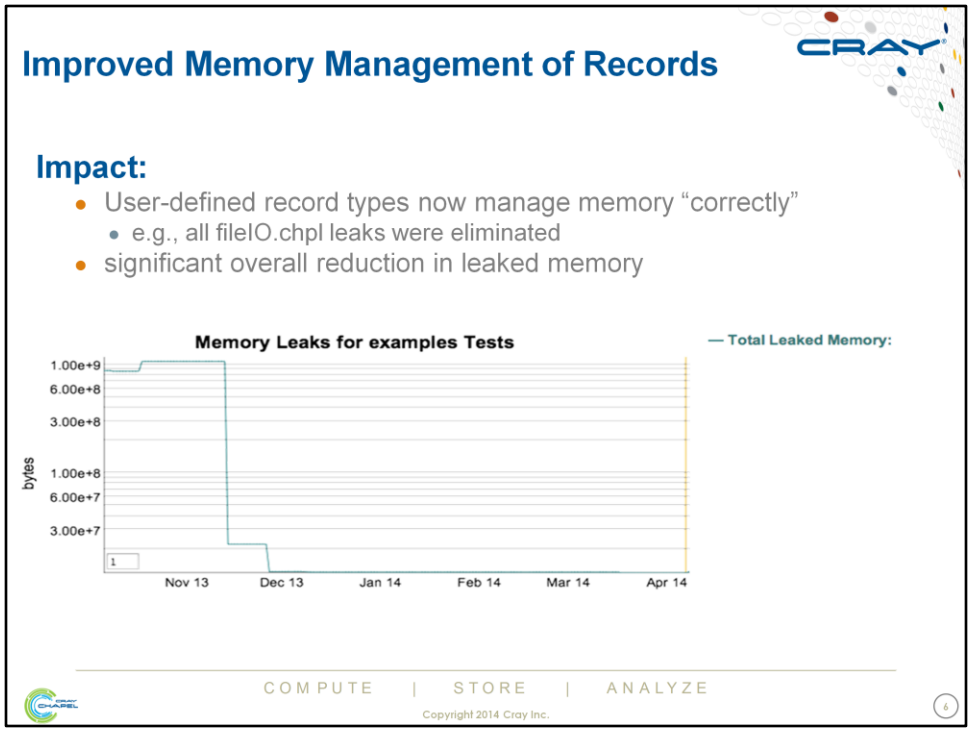
Outline

- Improved Memory Management of Records
- Improved Program Startup
- Improvements to Loop Invariant Code Motion
- Synchronization Variable Blocking Optimization
- Optimized Serialization of Data Parallelism
- Nested Parallelism Optimization
- Improved Task Allocation for Block Distribution
- Whole-array Binary I/O
- Cleanup of noRefCount-related Module Code
- Global Constant Replication Improvements
- Removing Unnecessary Formal Temps
- Improved const-ness, remote value forwarding
- Simplified Homogeneous Tuple Ops
- Copy Propagation Reimplemented


Improved Memory Management of Records
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
4



- Chapel's domain and array types are an example of an internal record type that was unmodified by this change; today they are treated specially. Over time, the goal is to treat them increasingly less specially.
- Other examples of internal record types include sync/single variables



- It should be noted that the memory leaks graph shown here does not track leaks of strings (all Chapel strings are currently leaked). A current effort is converting Chapel strings to records to make them less a special case and to benefit from this work.

Improved Memory Management of Records

- **Defining Chapel's User-Managed Memory (UMM) Model**
  - Closely follows the C++ memory-management model
  - Relies on "hooks" that allow user code to perform memory-management tasks
  - Default-constructor, copy-constructor, assignment-operator, destructor
    - Compiler supplies default implementations for these functions

- **Updated the compiler to insert all hook calls correctly**
  - Supports reference-counting and other MM schemes
  - Required only for user-declared variables and fields (not temporaries)
  - 'const ref' optimization is applied where appropriate

Improved Memory Management of Records

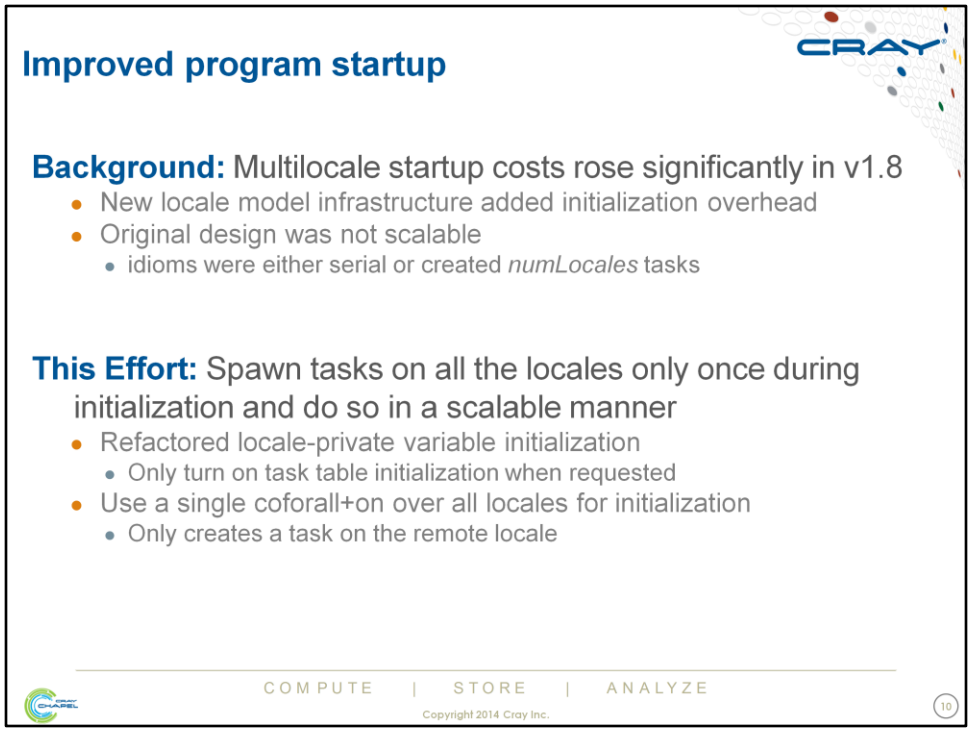
Next Steps:

- Enable a more natural coding style
  - Call user overrides of default constructors
  - Rename "initCopy" using the constructor naming convention.
- Propagate changes to internally-defined record types
  - This should drive remaining leaks to zero.

Future Work:

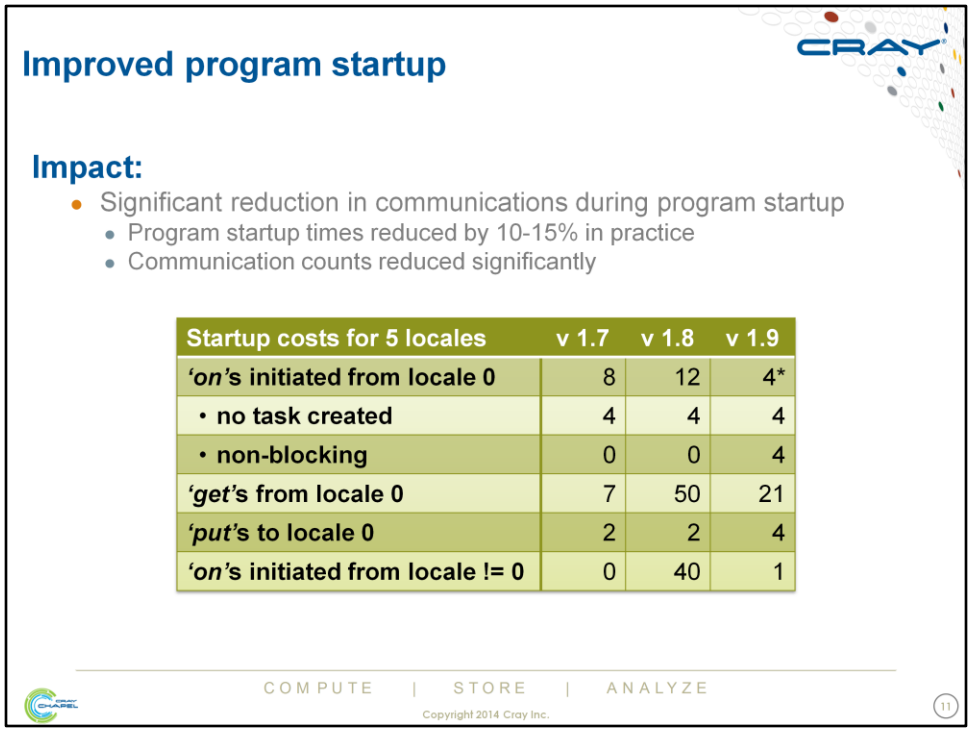
- Implement the new constructor model
  - Support calling constructors as "ordinary functions"
  - Avoid double-initialization
  - Eliminate the need for an initialize() function
- Implement rvalue-reference argument intents and binding
  - Support move-constructor, move-assignment and similar memory reuse

Improved Program Startup



- The original startup design tried to avoid requiring 2 tasks and possibly 2 threads per locale, which is why it avoided using a coforall+on, as that results in an optimization that creates a single non-blocking task on each of the remote locales, all of which must synchronize with locale 0, thereby requiring an additional task. Unfortunately, under normal operation, this would result in numLocales tasks being created on locale#0.

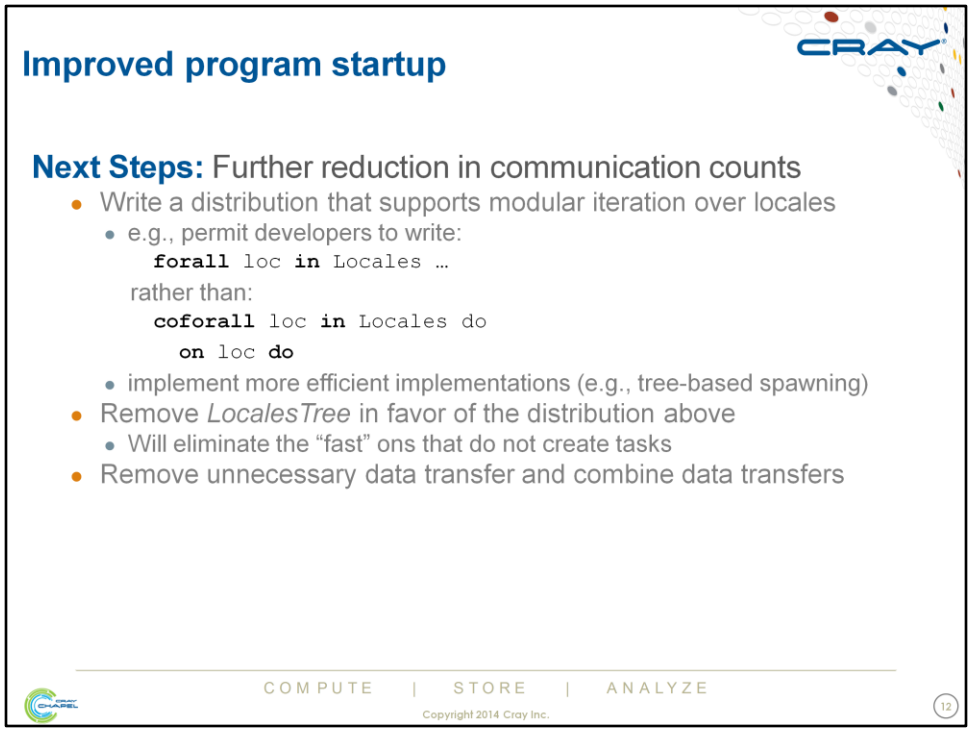
- This series of commits combined defaultDist and rootLocale initialization into the initOnLocales iterator, moved memory tracking initialization into the runtime, and disabled task table initialization unless requested explicitly. We also decided to relax the minimum tasks requirement and use a coforall+on to initialize data on all the locales.



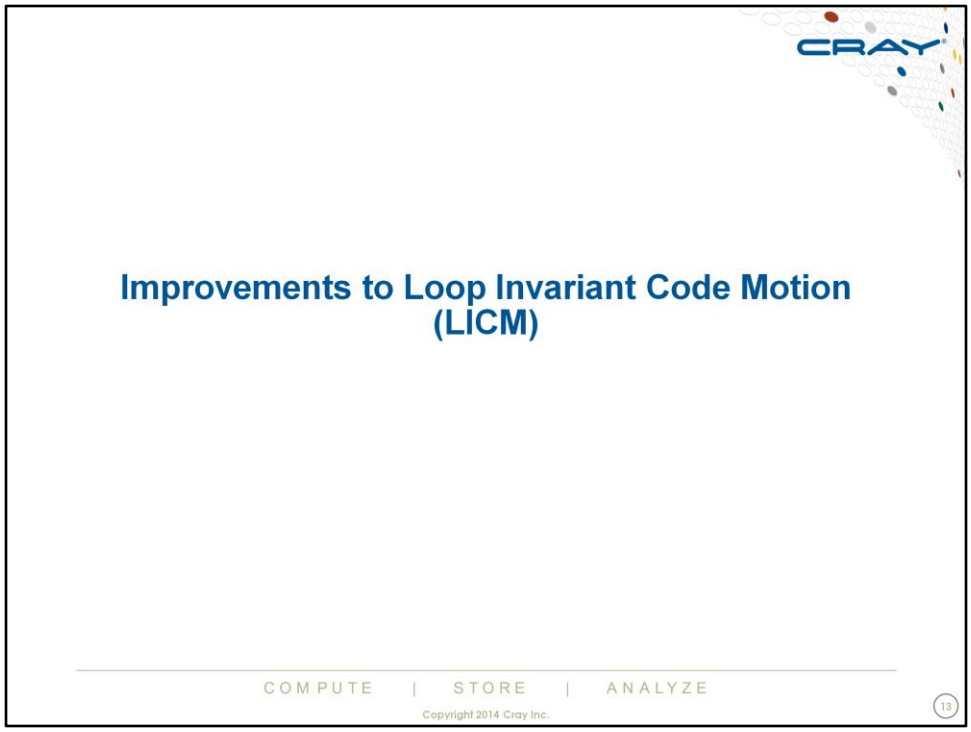
A nightly test was added to track start-up communication counts.

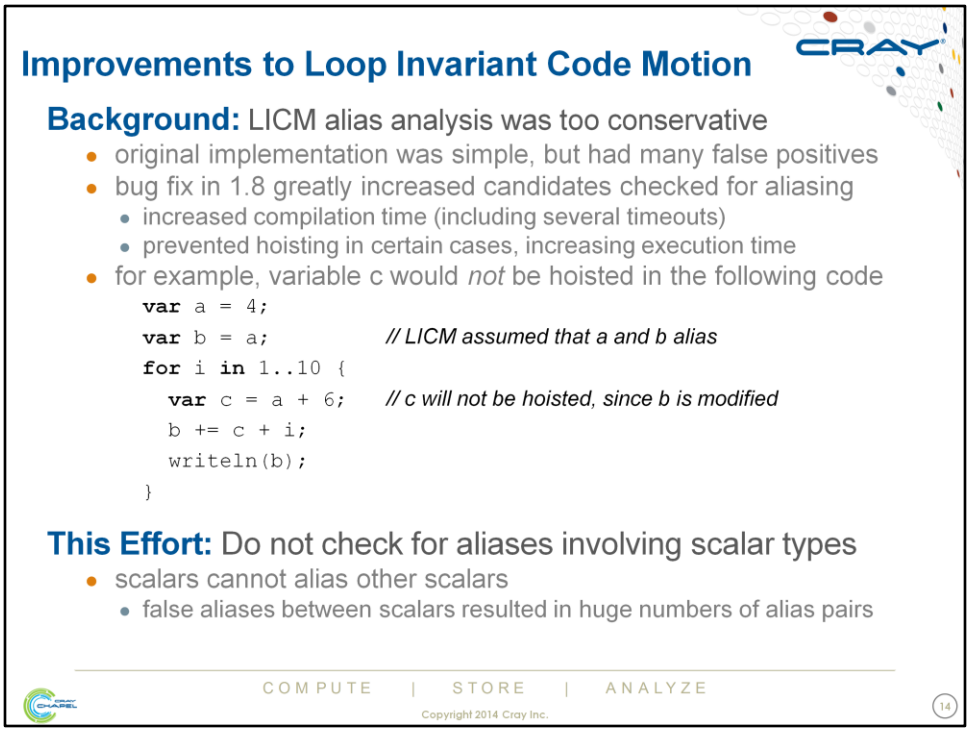
Details regarding communication counts:

- on from locale 0: barrier release (* for atomic update, would go away with AMOs for ugni comm layer)
- on no task created: LocaleTree initialization
- on non-blocking: coforall+on for defaultDist and rootLocale initialization
- get from locale 0: data transfer
- put to locale 0: data transfer
- on from locale != 0: join for coforall+on



The 'LocalesTree' concept was an early, non-modularized attempt to create tree-based parallelism across locales


CRAY
Improvements to Loop Invariant Code Motion (LICM)
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
13



The original alias analysis implementation existed because it was the simplest implementation that would detect all true aliases, but led to a lot of false positives

The intention has always been to update and improve it, but good alias analysis is a large undertaking

The original algorithm would basically search for any assignments and assume that every assignment creates an alias pair

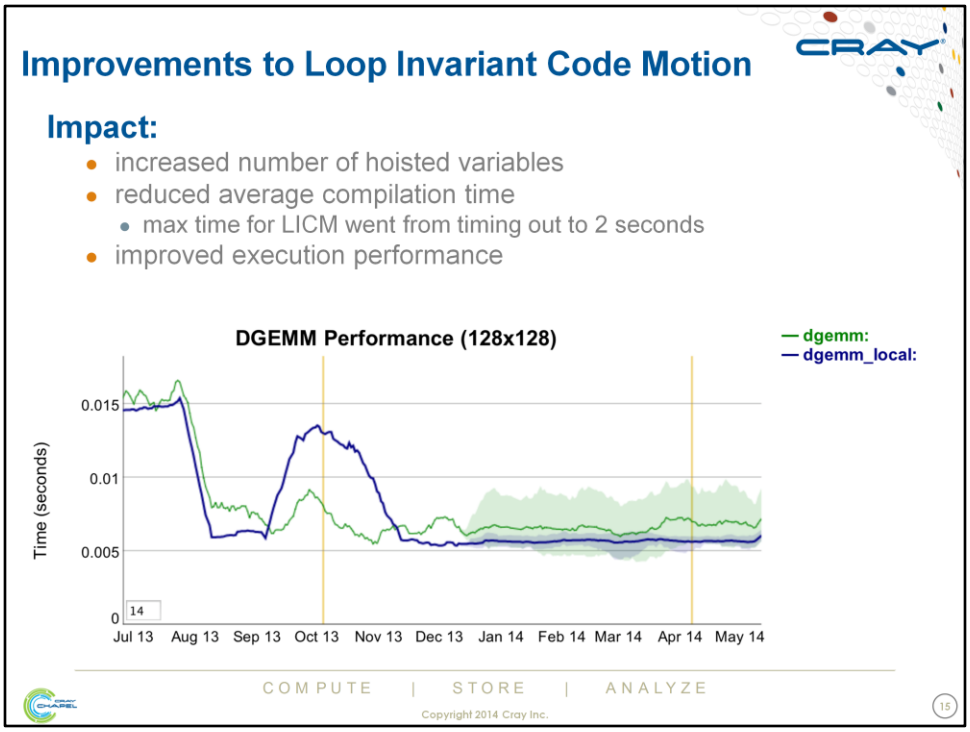
- This wasn't too bad because we were originally only checking variables inside of loops for aliasing, but this was wrong as aliases between variables in the loop can be created outside of the loop.

- r21811 corrected this issue, but led to a drastic increase in the number of aliases which led to higher compilation times and in some cases (like DGEMM) execution performance drops

Compilation timeouts were patched with a threshold for number of aliases in r21860 but this was only a stopgap solution

The fact that aliases were detected for scalars didn't initially have a large impact

because most of our performance benefits stemmed from hoisting array metadata, and that metadata is not affected by this overly conservative alias analysis. The update from r21811 made it the case that for dgemm and programs like dgemm this overly conservative analysis became a problem

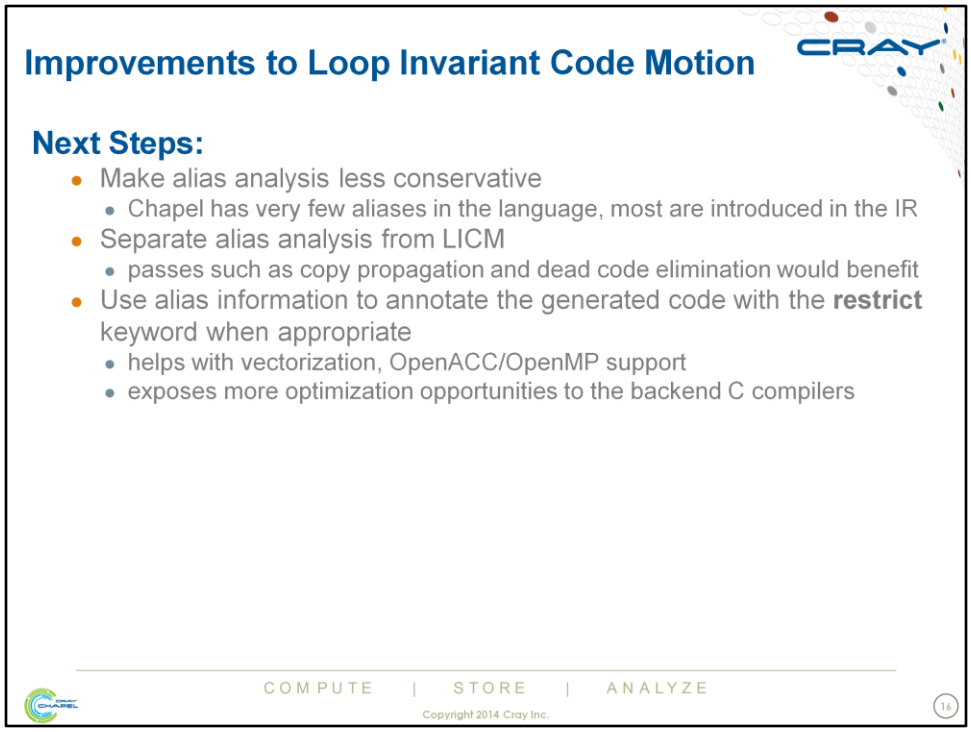


Patch to fix conservative alias analysis was r22228 on 10/30/2013

Performance impact was mostly seen for DGEMM (128x128 and 64x64)

Graph is shown with a roll period of 14 days because DGEMM is a noisy test. The important segments are that we had a steep performance benefit with LICM implementation seen before 1.9 release, and a decrease for DGEMM local close to the release, followed by a return to the all time low with the update to alias analysis.

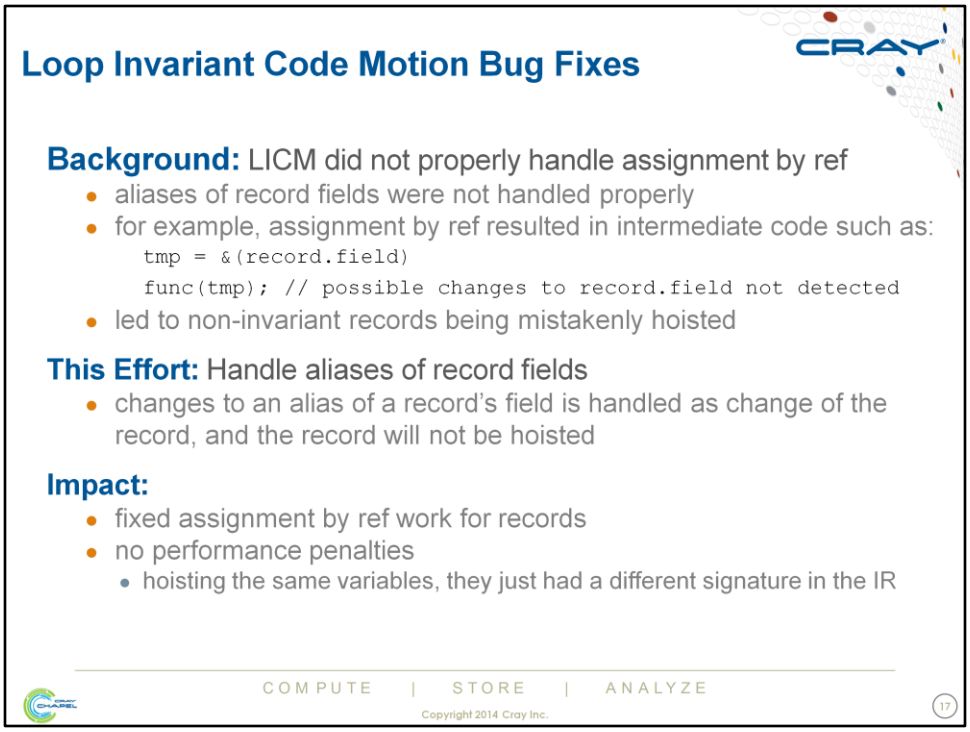
DGEMM is over twice as fast following addition LICM and alias updates, then prior to it.



Chapel has limited aliases in the language and most are added in the IR, so we should be able to track alias creation and thus determine only true aliases as they are created

- this would be significantly more accurate than the after the fact searching through assignments that we do now

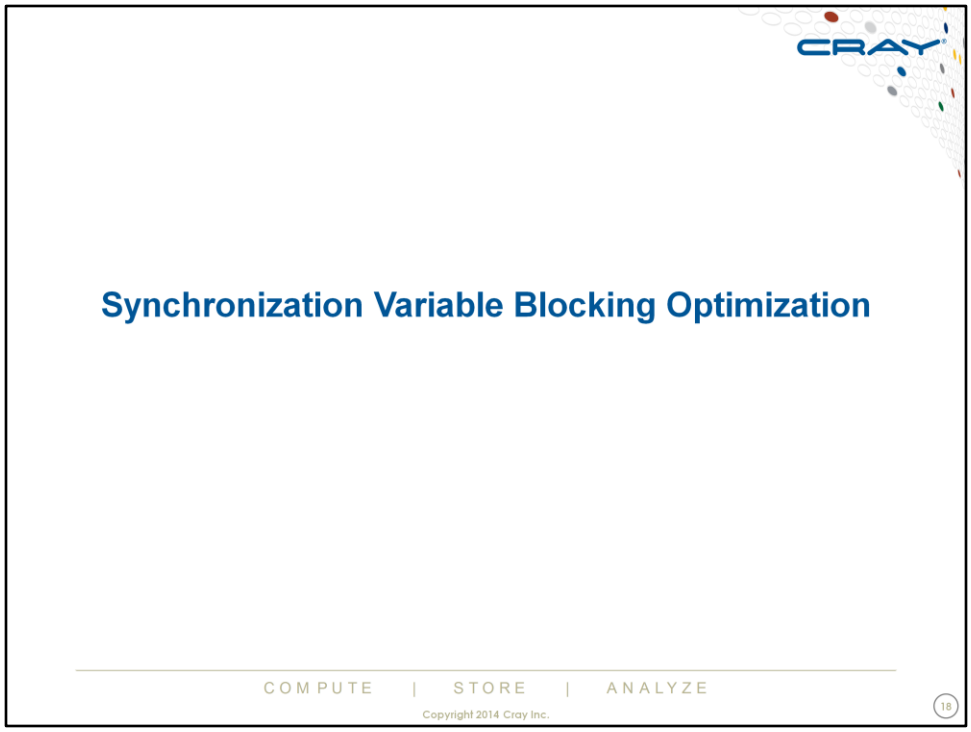
Letting the backend C compiler know when variables don't alias will allow it to generate more optimized instructions, and is important for performance for OpenACC/OpenMP as well as creating code that can be vectorized.



This bug was fixed with r22606 on 01/30/2014 and r22826 on 03/06/14

This bug was not exposed previously, it was only with the assignment by ref work that the IR had a form that made this bug hoist inappropriate expressions

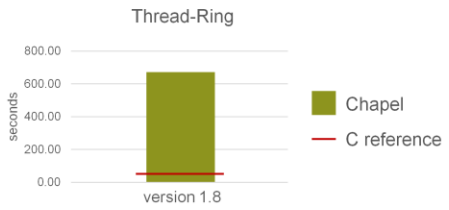
r22826 fixed a related bug for tuples. Tuples are implemented as records, but there is a special primitive to access their fields that was not checked.


Synchronization Variable Blocking Optimization
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
18

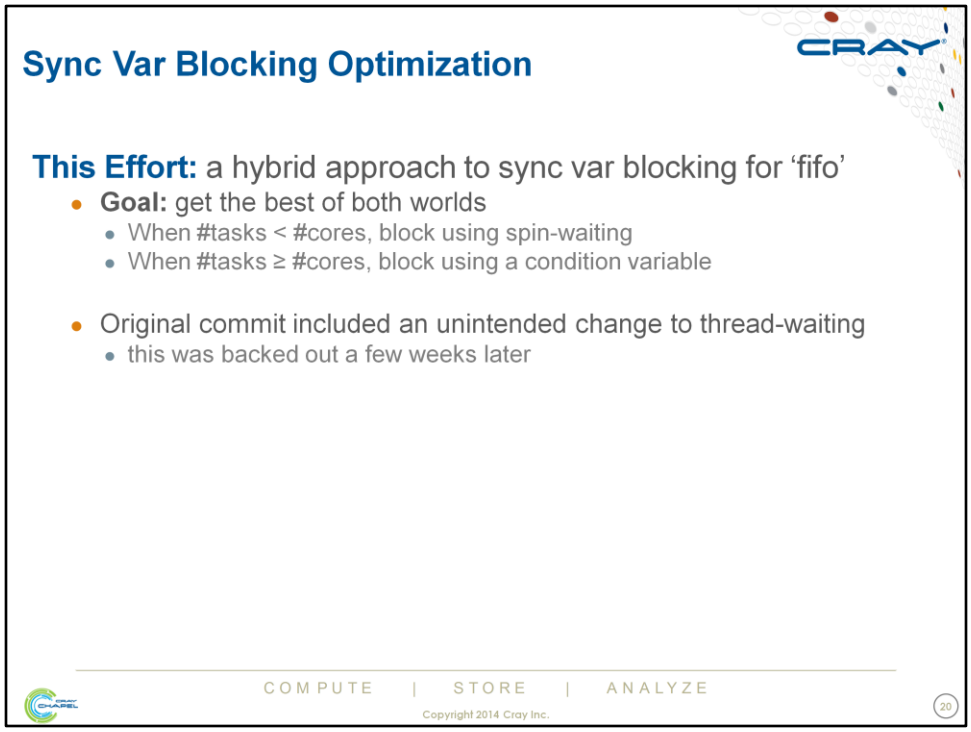
Sync Var Blocking Optimization

Motivation:

- Oversubscribed multitasking tends to perform poorly relative to C
  - Results vary with different tasking layers
    - 'fifo' (the default tasking layer) was particularly bad
    - for example, thread-ring was ~11x worse than the reference version
- The 'fifo' implementation of sync/single variables was a major problem
  - in most recent versions, threads have blocked using spin-waiting
    - previously, they had blocked using condition variables, but we switched to spin-waiting to reduce overhead for the common case when #tasks ≈ #cores



The current spin-waiting based approach is reasonable when the number of tasks is similar to the number of cores available. However, when there are a large number of tasks (thread-ring's measured problem size requires ~500), it is better to use condition variables.

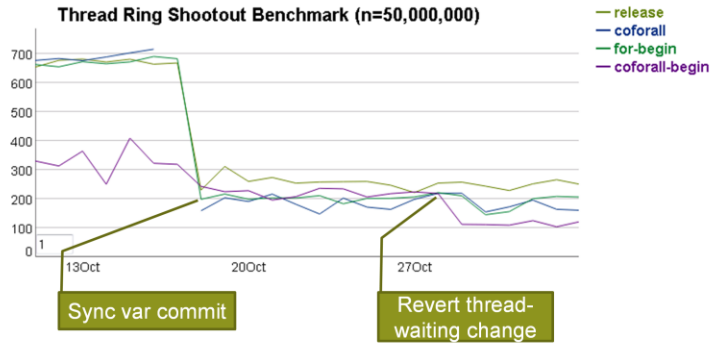


- The unplanned change was that threads, when waiting for a new task to run, inherited a similar hybrid scheme (historically, they have spin-waited as well for similar rationale; and do so today as well)
- Such a hybrid scheme could ultimately be useful, but in this implementation, we were worried about the possibility that threads would never get woken up in certain cases; and since this change wasn't intended, we backed it out
- If 'fifo' persists as a major tasking layer, this hybrid thread blocking strategy should be revisited; rather than spend more time on it, we decided to focus on Qthreads-based improvements instead

Sync Var Blocking Optimization

Impact (on thread-ring):

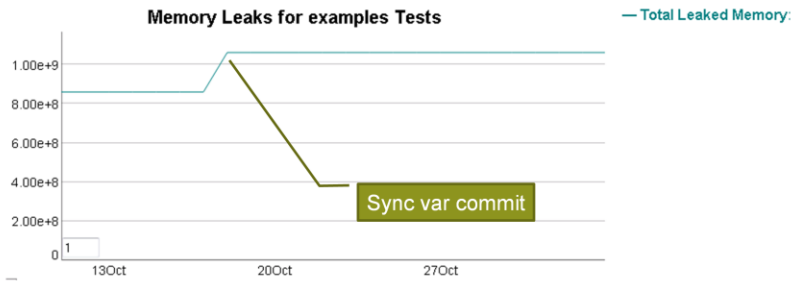
- Significantly improved thread-ring
  - ~2x faster than previous runs, only ~5x slower than reference
  - removed most disparities between different parallelization approaches



Sync Var Blocking Optimization

Impact (on memory):

- Chapel leaks sync variables in some situations
- Storing the condition variable used by this change made them larger
- Thus, this change increased the amount of leaked memory

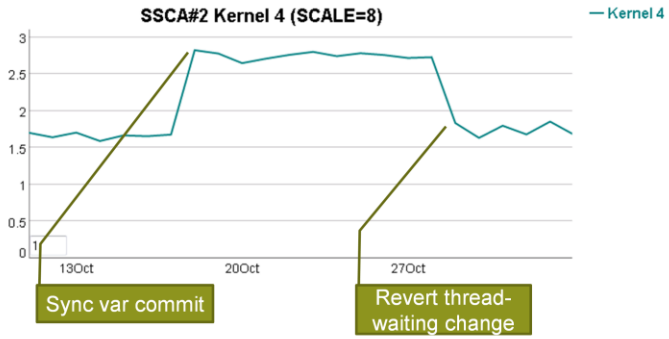


Some reclamation of this memory has occurred, but not all of it.

Sync Var Blocking Optimization

Impact (on other benchmarks)**:**

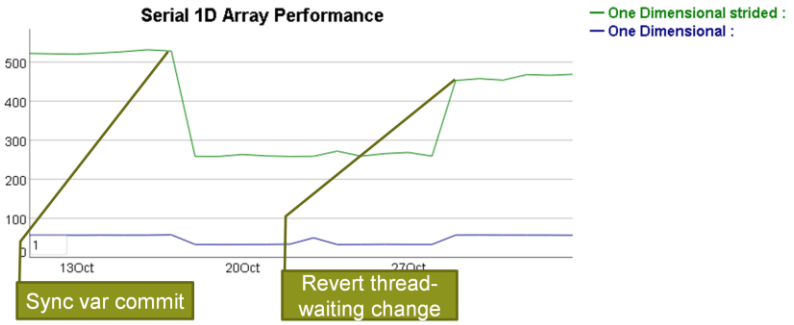
- Several tests were impacted negatively by the thread-waiting change
  - e.g., SSCA#2, LULESH, spectral-norm
- When that change was reverted, so was the performance regression



Sync Var Blocking Optimization

Impact (on other benchmarks):

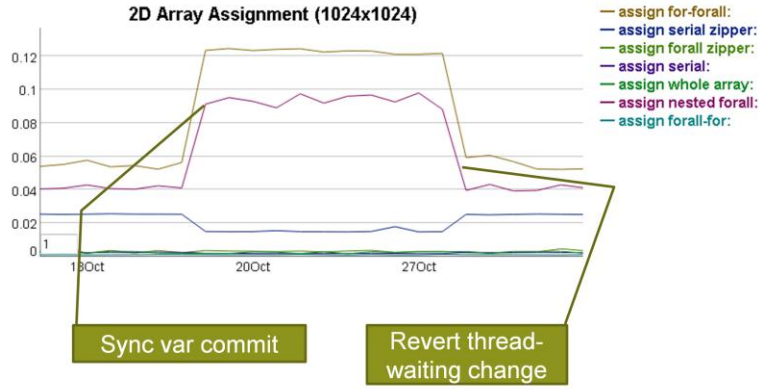
- Other tests benefitted from the thread-waiting change
  - e.g., Mandelbrot, serial 1D array performance, promoted op=
- When that change was reverted, so was much of the benefit



Sync Var Blocking Optimization

Impact (on other benchmarks):

- This test shows that a single type of computation had both gains and losses from these changes depending on how it was written



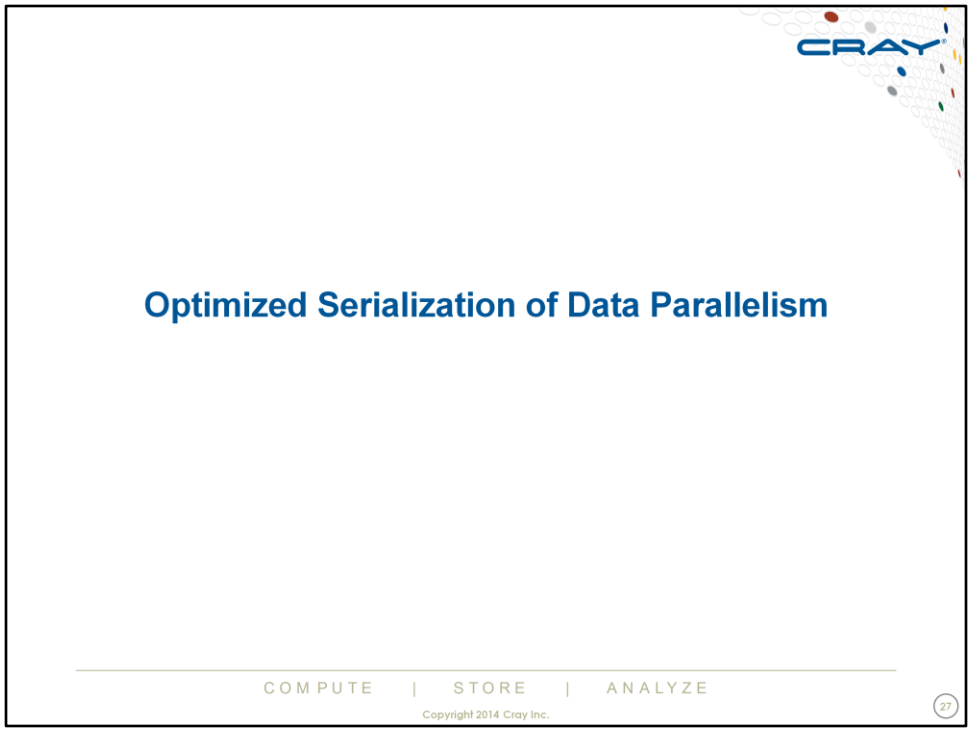
Sync Var Blocking Optimization

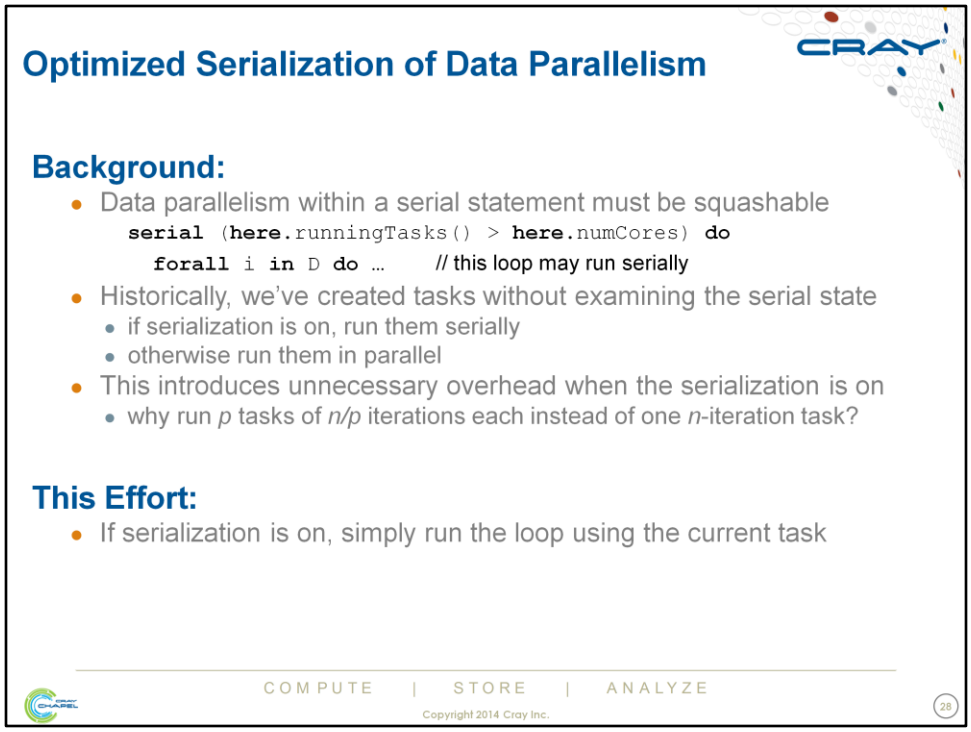
Next Steps:

- Work toward making Qthreads the default
  - should result in performance improvements for benchmarks like thread-ring
  - ideally without performance regressions for other tests
- Close sync variable memory leaks

Future Work:

- Consider re-implementing hybrid thread-waiting strategy
  - assuming 'fifo' continues to be used/useful


Optimized Serialization of Data Parallelism
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
27

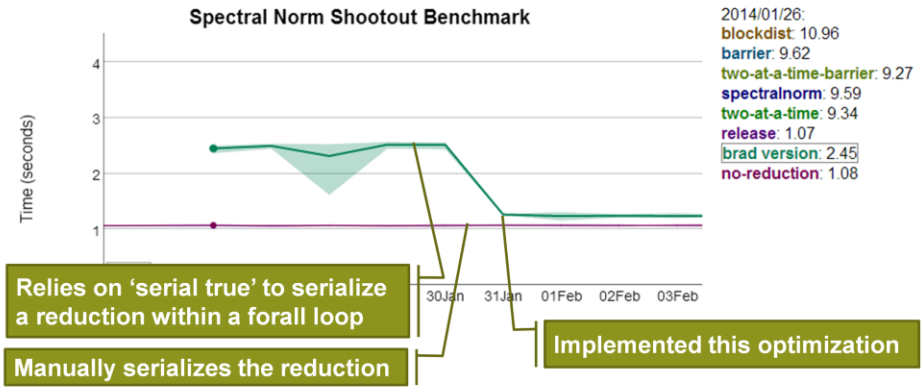


Note that while in some cases you may know statically that you will want to execute a loop in parallel and can replace the forall with a for, in constructs where we use foralls under the covers, such as reductions, we need to use an serial statement.

Optimized Serialization of Data Parallelism

Impact:

- Idioms relying on serial statements to throttle nested parallelism saw significant improvement

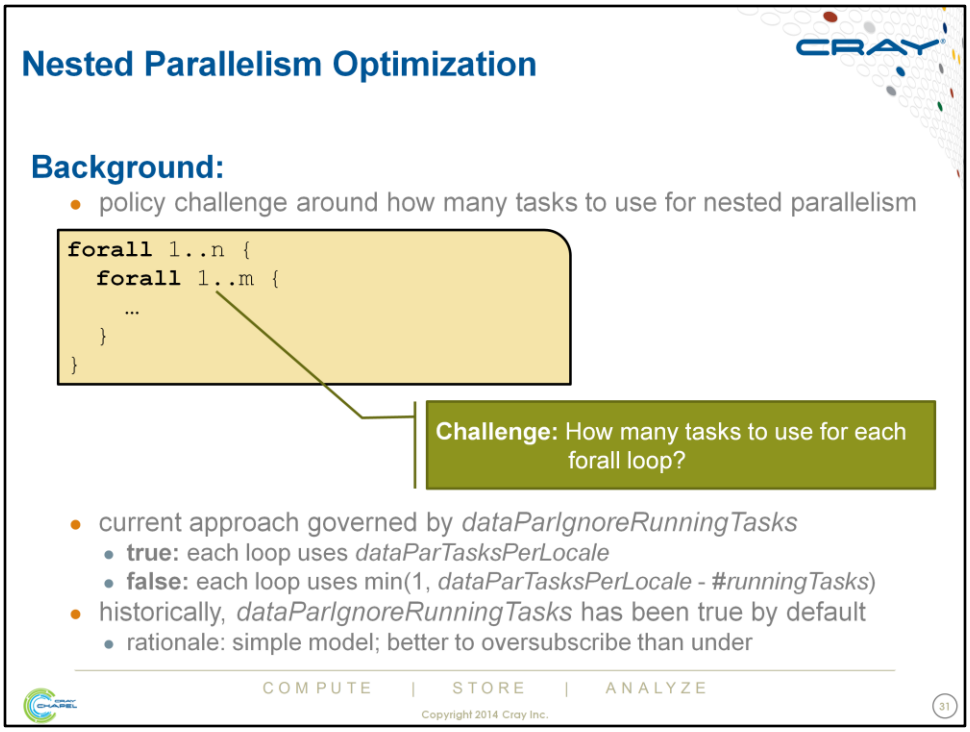


Relies on 'serial true' to serialize a reduction within a forall loop

Manually serializes the reduction

Implemented this optimization

Nested Parallelism Optimization



- "uses" here is an approximation to reality, because it's also governed by dataParMinGranularity which can cause fewer tasks to be used…
- clearly "better to oversubscribe than under" isn't always true, but it felt like the less risky proposition originally…

Nested Parallelism Optimization

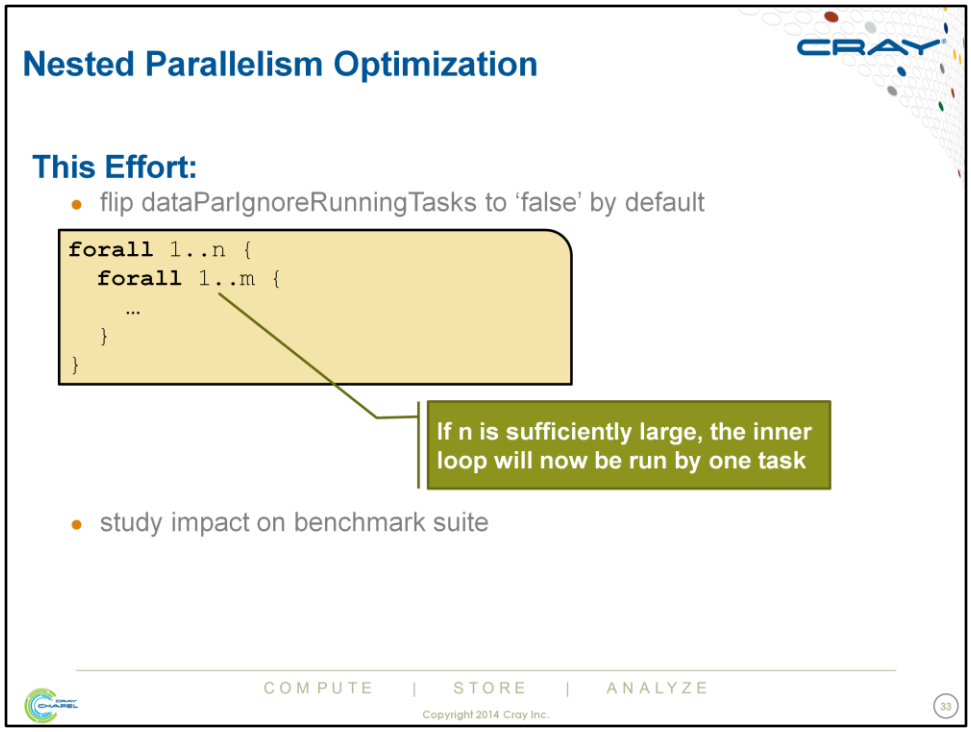
Motivation:

- too much parallelism

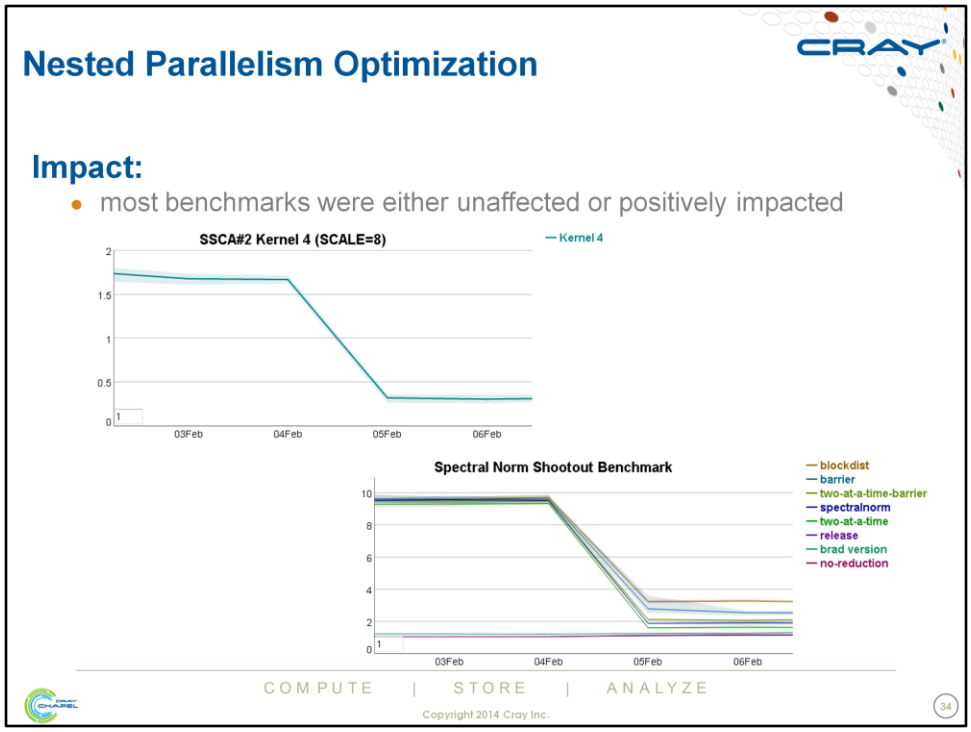
```
forall 1..n {
  forall 1..m {
    …
  }
}
```

Cases like this could easily create (numCores)^d tasks, for a *d*-nested loop

- the impact became more dramatic as core counts have increased



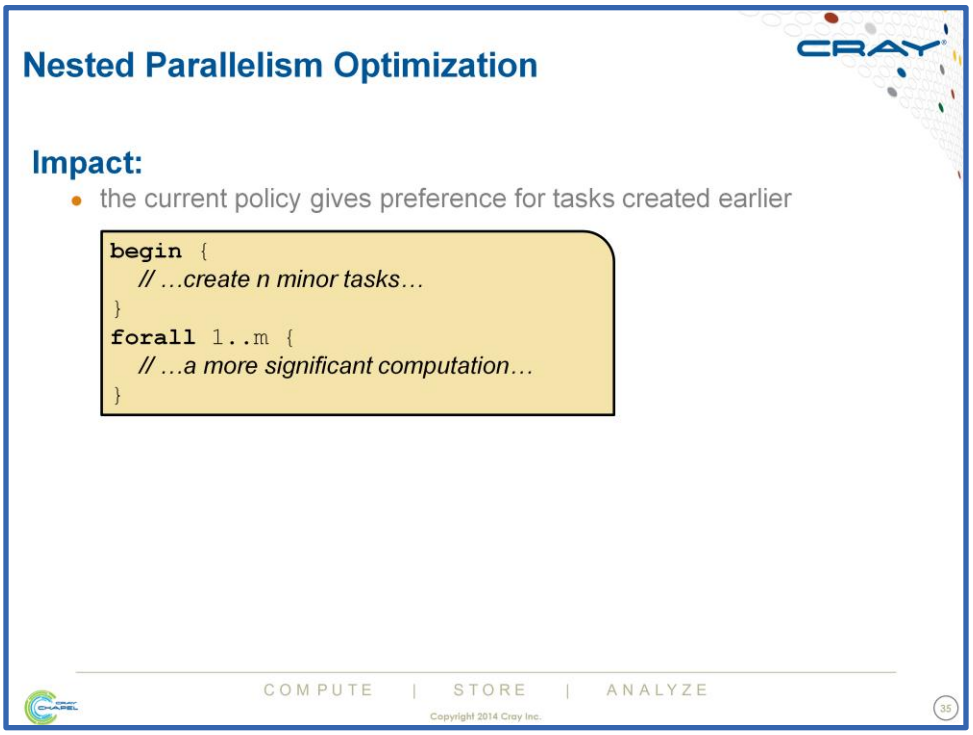
"If n is sufficiently large" means "results in enough tasks to fill all the processor cores", as governed by dataParMinGranularity



- This generated a record time for SSCA#2 on our nightly performance testing (which has since been beaten again)

- Most spectral-norm versions made use of reductions within forall statements (nested parallelism) and so improved with this change; the flat lines are versions in which the reduction was serialized and was therefore unaffected by this change.

- Additionally, it was discovered that coforall + on created an additional task on the starting locale because it does not discount itself when checking the number of tasks created. This did not dramatically affect performance when we were ignoring running tasks, but when the count mattered it became an issue for Block and Cyclic distributions, specifically. The next set of slides will address this issue.



- In this case, should the “begun” tasks last long enough to affect the forall loop, the program will assume it cannot make use of as many tasks and so will not apply as many tasks to the forall as it would were dataParIgnore flipped.
- Of course, users can always override the default, globally or on a loop-by-loop basis

Nested Parallelism Optimization

Next Steps:

- explore additional mechanisms for optimizing task creation
  - compiler analysis
  - syntactic/semantic constructs
  - runtime throttling
  - inspector/executor strategies
  - profile-guided optimization
  - …

Improved Task Allocation for Block Distribution

Improved Task Allocation for Block Distribution

Background:

- The Block distribution's iterators use the running task count to determine the number of tasks to create on each locale
  - This count includes the task that initiated the loop if it does not block before the query is made

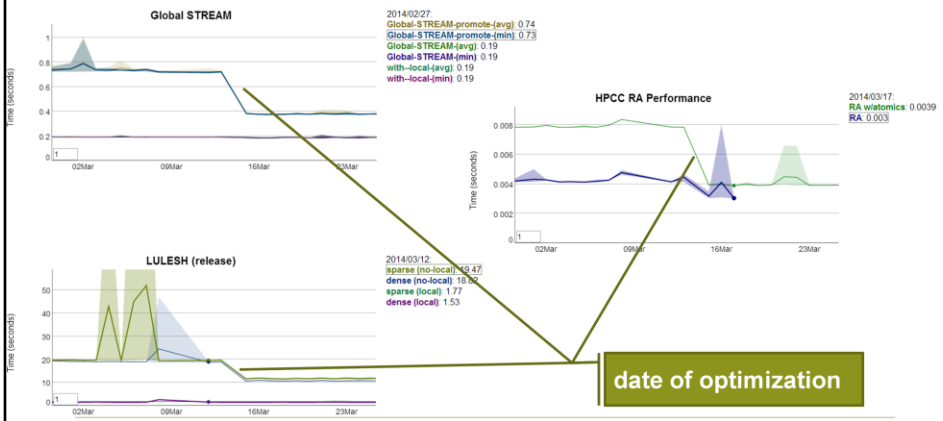
This Effort:

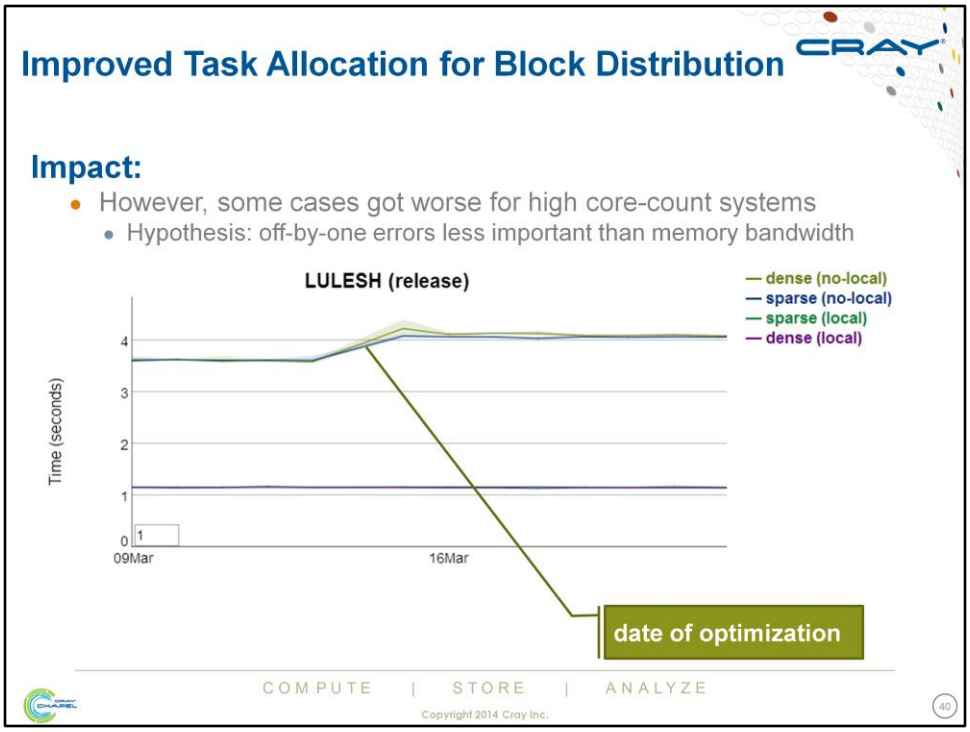
- Ignore the initiating task if it is the only one on the locale
  - If the initiating task is the only one, we are executing in a purely data parallel manner
  - If there are other tasks, we are either using nested or unstructured parallelism, so count all tasks in an attempt to be more fair in task allocation

Improved Task Allocation for Block Distribution

Impact:

- All available cores on the locale of the initiating task are being used
- Benchmarks using the Block distribution improved for low core-counts





A NUMA-aware locale model may potentially help mitigate these overheads. In particular, these performance numbers are currently being gathered in a NUMA-oblivious manner.

Improved Task Allocation for Block Distribution

Next Steps:

- Improved tasking heuristics to get the best of both worlds
  - or, more generally, how to account for memory in choosing task counts?
- Implement a more general solution to the task counting problem
  - e.g., provide a way for tasks to request to *not* be counted in such cases

Whole-array Binary I/O

Whole-Array Binary I/O

Background: Historically, array I/O has taken many operations

- Each element results in one call

- The extra reads and writes add up quickly
- It would be nice to merge them when possible

Whole-Array Binary I/O

This Effort: Binary I/O is now done with just one operation

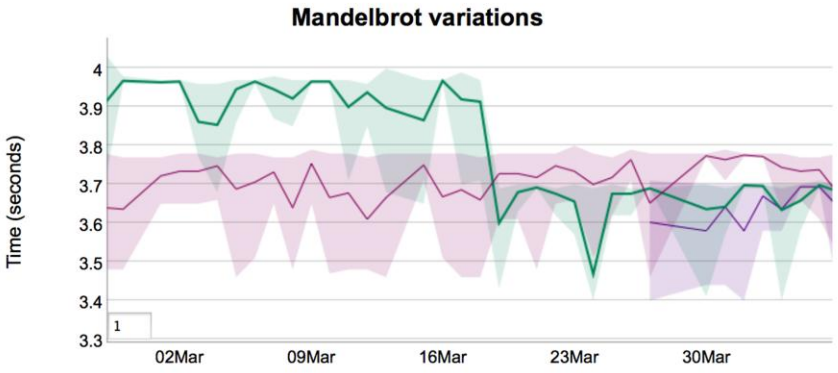
- The whole array is read in or written out at once

- This only happens when several conditions are true:
  - Binary I/O
  - Simple element types (int, uint, real, complex, …)
  - Byte order of the operations matches the system byte order
  - The array must be DefaultRectangular

Whole-Array Binary I/O

Sample Impact: Mandelbrot speedup

- The test in green below gets a ~12% speedup with whole-array I/O

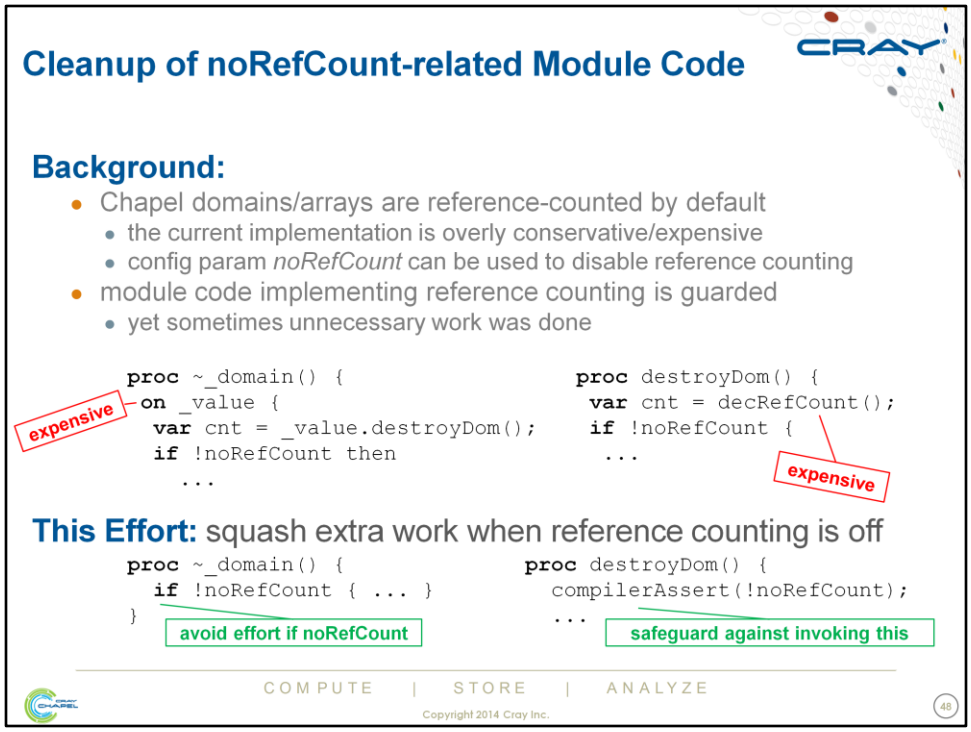


Whole-Array Binary I/O

Future Work: Relax the conditions for this capability

- Reads can work when the byte order does not match
- Add support for types composed of simple types (records, tuples)
- Add a limited form of batched writes for other distributions

Cleanup of noRefCount-related Module Code

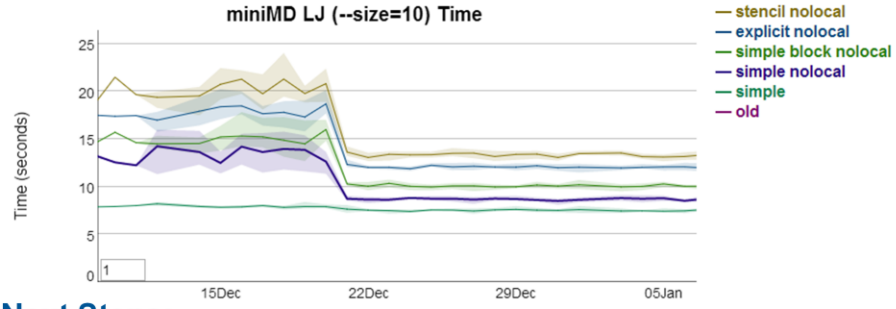


Reference counting is important because it enables reclamation of arrays and domains that are no longer used. Currently the overhead of reference counting can be significant. Setting noRefCount=true is a workaround for performance; it can also result in excessive memory use.

Commit 22473.

Cleanup of noRefCount-related Module Code

Impact: ~30% speedup on miniMD benchmarks

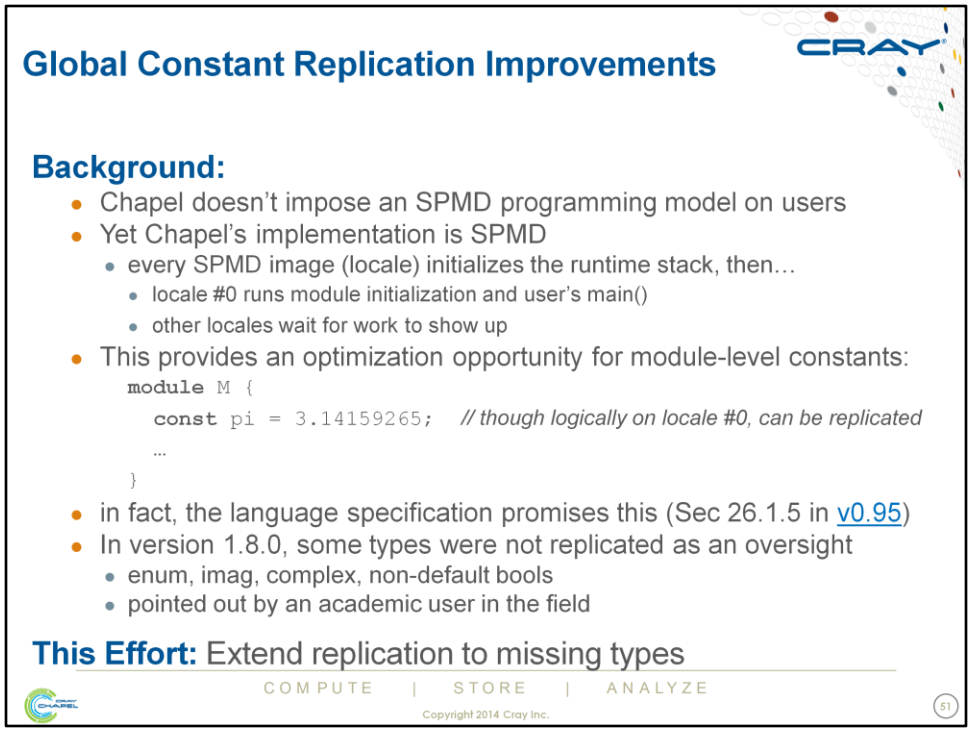


Next Steps:

- reduce the cost of reference counting for default compilations
  - decouple reference counting from argument passing semantics
  - optimize away reference counts for cases that don't require them
- deprecate the 'noRefCount' config param

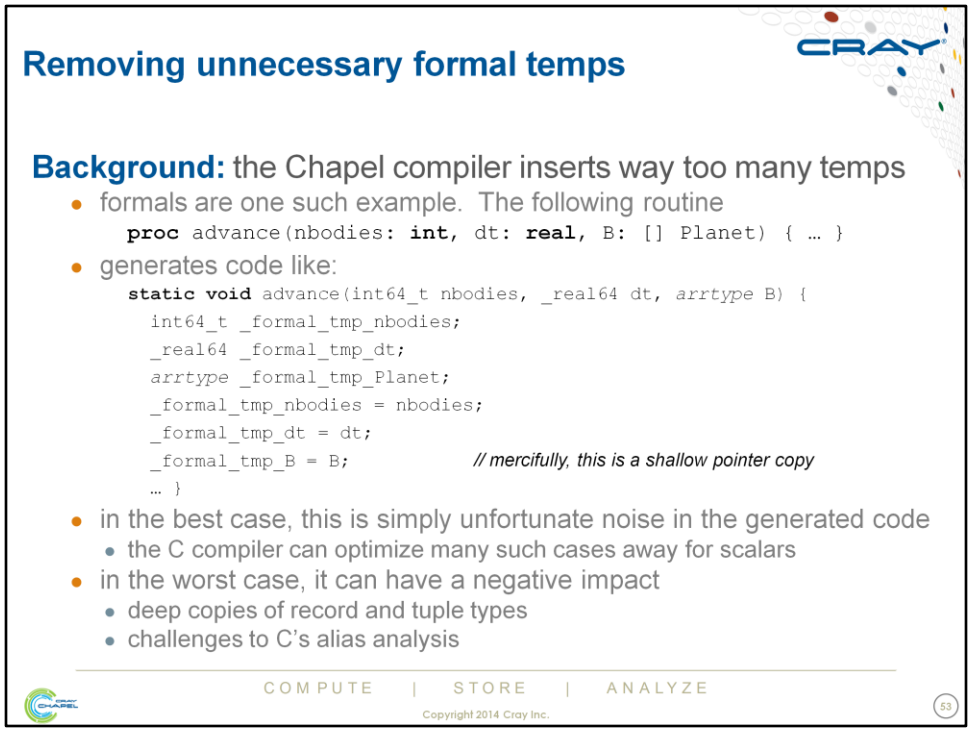
Note: miniMD benefits so much from this because it throws the –snoRefCount flag. Eventually, when the flag is deprecated, the goal would be to get similar performance without using a special flag.

Global Constant Replication Improvements

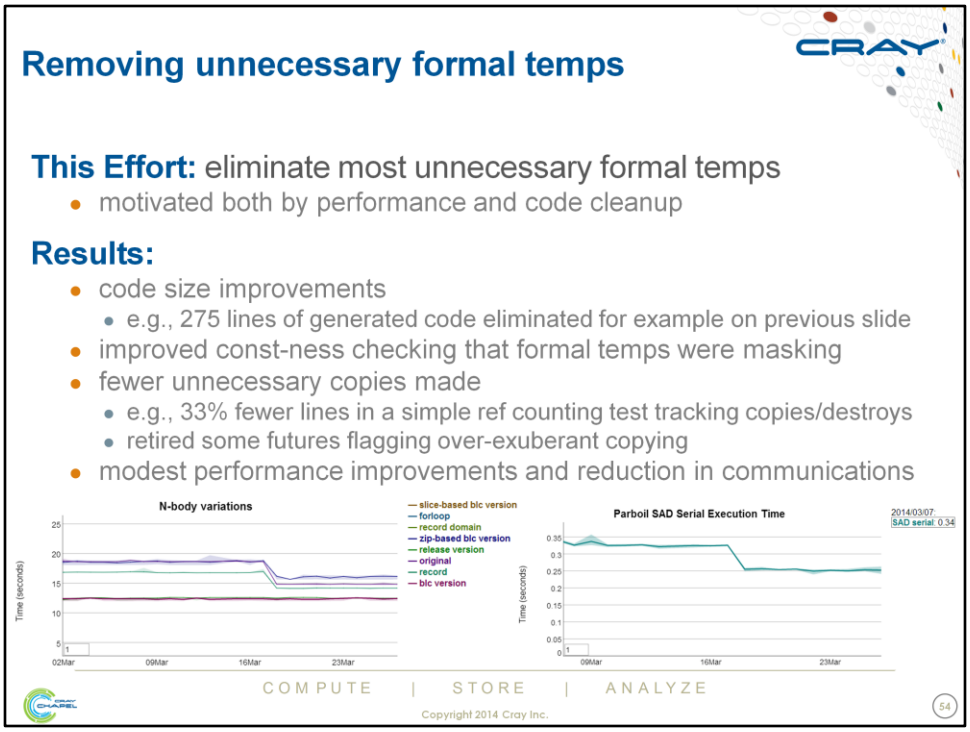


- 'pi' logically lives on locale #0; but since it's invariant, each locale can store its own local copy
- Though this change can potentially have significant performance effects for such cases, we didn't see an impact in our automated testing due to lack of multi-locale performance testing as well as cases that rely on such constants

Removing Unnecessary Formal Temps



- The historical reason for inserting formal temps conservatively (and temps in general) is to favor correctness/simplicity in the implementation first, performance afterwards. The time for performance has come, ideally without hurting simplicity (in this case, that was achieved and the result was simpler logic in the compiler).
- This code sample comes from one of our n-body codes from the Computer Language Benchmark Game: test/studies/shootout/nbody/sidelnik/nbody_orig_1.chpl.
- arrType is a placeholder for the much longer type name used by the Chapel compiler for this array



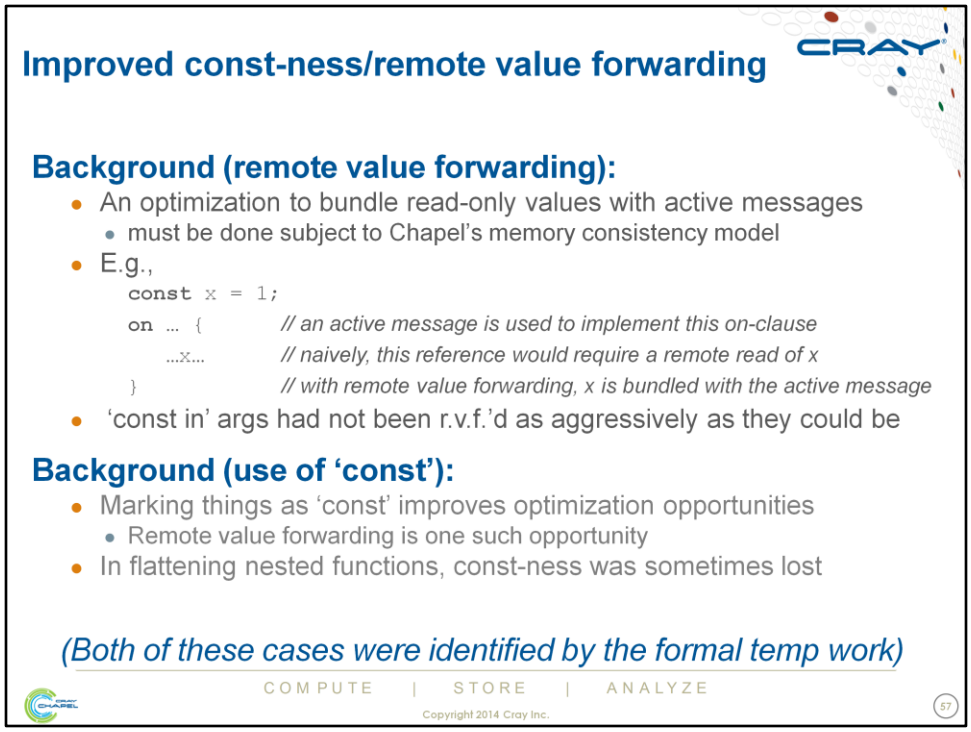
- This was r22900 (and a few follow-up ones w.r.t. testing), made on March 17, 2014
- Examples of tests for which formal temps had masked illegal assignments to const records: test/functions/deitz/test_formal_copy1.chpl, test/nostdlib/recordpass6.chpl, test/nostdlib/recordpass7.chpl
- The reference counting test in question is test/users/ferguson/refcnt.chpl
- Bigger performance improvements arguably weren't seen in part because of a chicken-and-egg issue: we tend not to pass records around in benchmark codes because of the extra copies they've traditionally induced. For this reason, the reduction in output in the ref counting test is perhaps more telling than the performance improvements themselves.
- In the n-body graph, the light purple line marked "original" corresponds to the nbody code shown on the previous slide.

Removing unnecessary formal temps

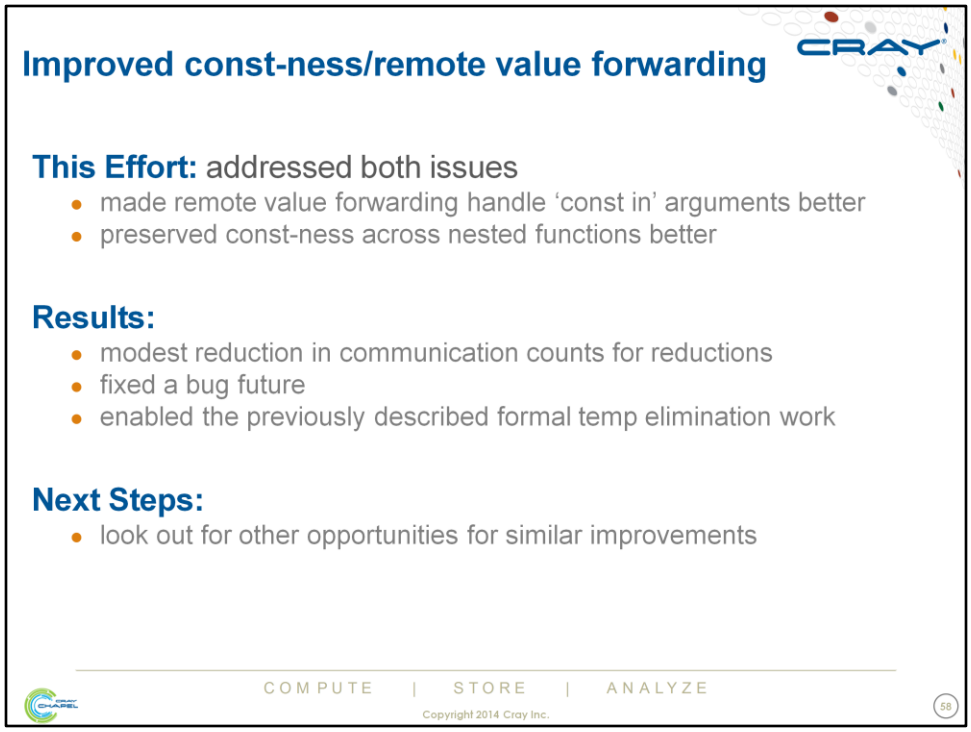
Next Steps: eliminate other unnecessary temporaries

- ref temps and call temps are two of the major cases remaining

Improved const-ness, remote value forwarding



- remote value forwarding = bundling read-only values with the active messages that implement on-clauses (subject to the memory consistency model)
- even if a user does not use nested functions, the compiler uses them frequently to implement task parallelism and on-clauses, so propagating const-ness across them is important



- This was r22894 (and a few follow-up ones w.r.t. testing), made on March 15, 2014
- The bug future in question was test/users/vass/crash4replLclsWFlds.future

Simplified Homogeneous Tuple Ops

Simplified Homogeneous Tuple Ops

Background:

- Chapel supports operators on tuples:

```
(1, 2, 3) + (4, 5, 6) => (5, 7, 9)
("hi", 1) + ("ya", 2) => ("hiya", 3)
```

- The traditional implementation of these operators has been recursive:

```
inline proc +(a: _tuple, b: _tuple) {
  if a.size != b.size then
    compilerError("tuple operands to + have different sizes");
  if a.size == 1 then
    return (a(1)+b(1),);
  else
    return (a(1)+b(1), (...chpl__tupleRest(a)+chpl__tupleRest(b)));
}
```

- Reason:
  - In the heterogeneous case, it's difficult to declare the result type a priori
  - Elegant, general Lisp-like

Simplified Homogeneous Tuple Ops

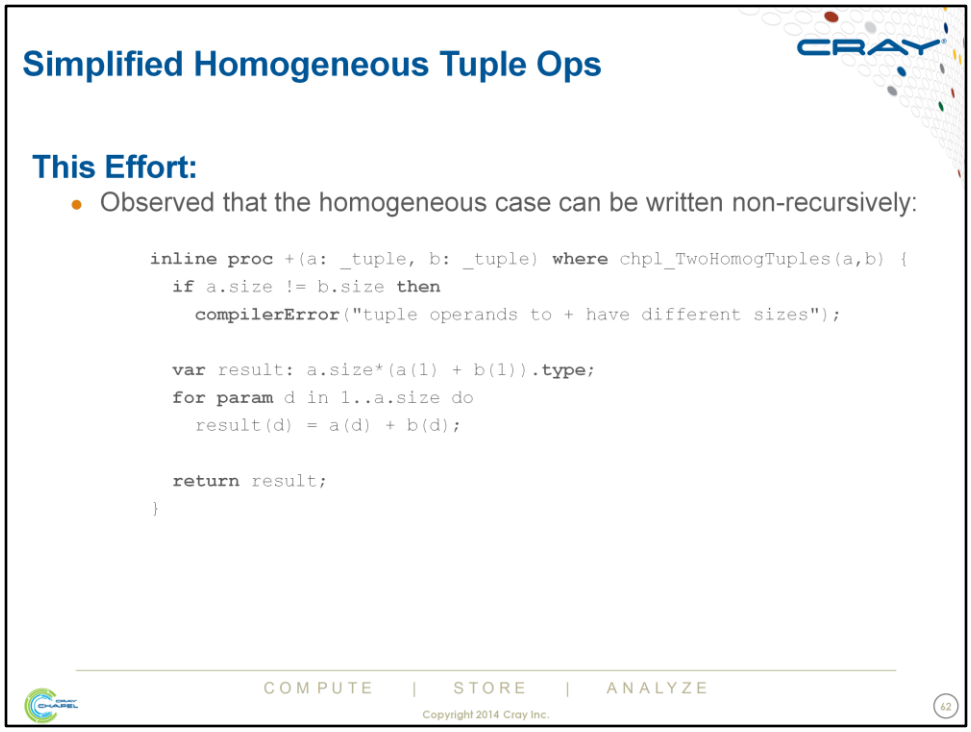
More Background:

- The generated code for this recursive approach is awful:

```
var x = (1.0, 2.0, 3.0),
    y = (4.0, 5.0, 6.0),
    z = x + y;
```

- Results in the following code:

```
call_tmp = (1.0 + 4.0);
ret_to_arg_ref_tmp_ = &wrap_call_tmp;
chpl__tupleRestHelper2(1.0, 2.0, 3.0, ret_to_arg_ref_tmp_);
ret_to_arg_ref_tmp_2 = &wrap_call_tmp2;
chpl__tupleRestHelper2(4.0, 5.0, 6.0, ret_to_arg_ref_tmp_2);
call_tmp2 = *(wrap_call_tmp + INT64(0));
call_tmp3 = *(wrap_call_tmp2 + INT64(0));
call_tmp4 = (call_tmp2 + call_tmp3);
call_tmp5 = *(wrap_call_tmp + INT64(0));
call_tmp6 = *(wrap_call_tmp + INT64(1));
ret_to_arg_ref_tmp_3 = &wrap_call_tmp3;
chpl__tupleRestHelper(call_tmp5, call_tmp6, ret_to_arg_ref_tmp_3);
call_tmp7 = *(wrap_call_tmp2 + INT64(0));
call_tmp8 = *(wrap_call_tmp2 + INT64(1));
ret_to_arg_ref_tmp_4 = &wrap_call_tmp4;
chpl__tupleRestHelper(call_tmp7, call_tmp8, ret_to_arg_ref_tmp_4);
call_tmp9 = *(wrap_call_tmp3 + INT64(0));
call_tmp10 = *(wrap_call_tmp4 + INT64(0));
call_tmp11 = (call_tmp9 + call_tmp10);
*(z + INT64(0)) = call_tmp;
*(z + INT64(1)) = call_tmp4;
*(z + INT64(2)) = call_tmp11;
```



- The reason this case can be done is because we can determine the type of a single component and then declare a homogeneous tuple of that type
- This commit was made in r22475, December 21st 2013

Simplified Homogeneous Tuple Ops

This Effort:

- Now we get much cleaner generated code:

```
var x = (1.0, 2.0, 3.0),
    y = (4.0, 5.0, 6.0),
    z = x + y;
```

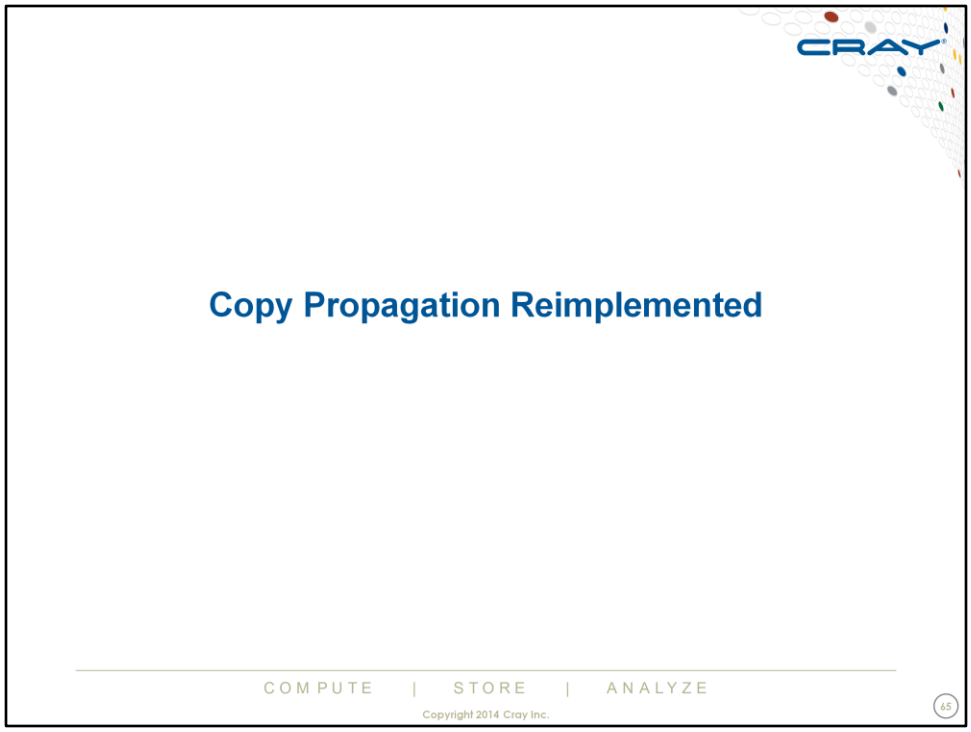
- Results in the following code:

```
call_tmp = (1.0 + 4.0);
call_tmp2 = (2.0 + 5.0);
call_tmp3 = (3.0 + 6.0);
*(z + INT64(0)) = call_tmp;
*(z + INT64(1)) = call_tmp2;
*(z + INT64(2)) = call_tmp3;
```

Simplified Homogeneous Tuple Ops

Next Steps:

- What language changes would enable a non-recursive approach for the heterogeneous case too?


Copy Propagation Reimplemented
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
65

Copy Propagation Reimplemented

- **The Problem:**
  - Old CP implementation did not track assignment through references
  - Instead, only killed candidate pairs when a reference was created
  - Had the potential to produce incorrect results

```
a = 1;         // Creates candidate pair (a, 1)
int* ap = &a; // Kills candidate pair
a = 2;         // New candidate pair (a, 2)
*a = 3;        // Ignored
b = a;         // Substitutes b = 2.  Oops!
```

  - But, not many references in older code, so this bug remained hidden
  - New assignment signature uses references more extensively
  - Uncovered a similar bug in Loop-Invariant Code Motion

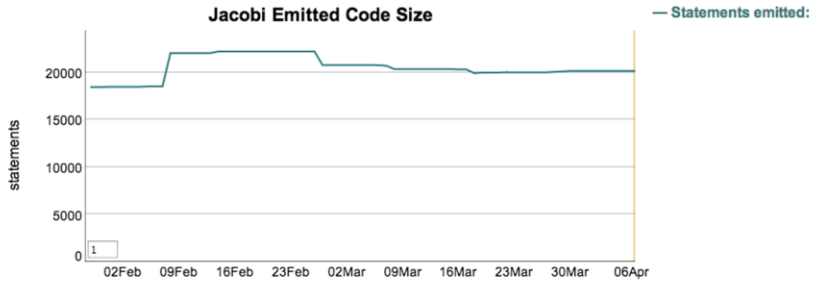
Copy Propagation Reimplemented

- **The Solution:**
  - New CP tracks references across an entire function
  - Candidate pairs are killed when the content of a reference is updated

```
a = 1;          // Creates candidate pair (a, 1)
int* ap = &a; // Noted in a previous pass
a = 2;          // New candidate pair (a, 2) kills (a, 1)
*a = 3;         // Kills candidate pair (a, 2)
b = a;          // No substitution
```

  - Bonus: We now propagate literals as well as constant values
  - 10% reduction in generated code size

Copy Propagation Reimplemented



- **Next Steps**
  - Track references dynamically
  - Track updates through references

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/