Benchmarks, Performance Optimizations, and Memory Leaks

Chapel Team, Cray Inc.

Chapel version 1.13

April 7, 2016

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline

- **Benchmark Improvements**
  - LCALS: Livermore Compiler Analysis Loop Suite
  - MiniMD Benchmark Improvements
  - The ISx Benchmark in Chapel
- **Performance Optimizations**
  - Bulk Transfer Improvements
  - Local On-Statements
  - Numa maxTaskPar Fix
  - Array-as-Vector Improvements
  - Reduction Performance Improvements
  - Performance of LLVM Back-End
  - Anonymous Counted Range Optimization
- **Memory Leak Improvements**
  - Lexical Scoping Improvements
  - Evaluation of Current Memory Leaks

LCALS: Livermore Compiler Analysis Loop Suite

LCALS: Background

- **LCALS: Livermore Compiler Analysis Loop Suite**
  - Loop kernels designed to measure compiler performance
  - Developed by LLNL
  - https://codesign.llnl.gov/LCALS.php

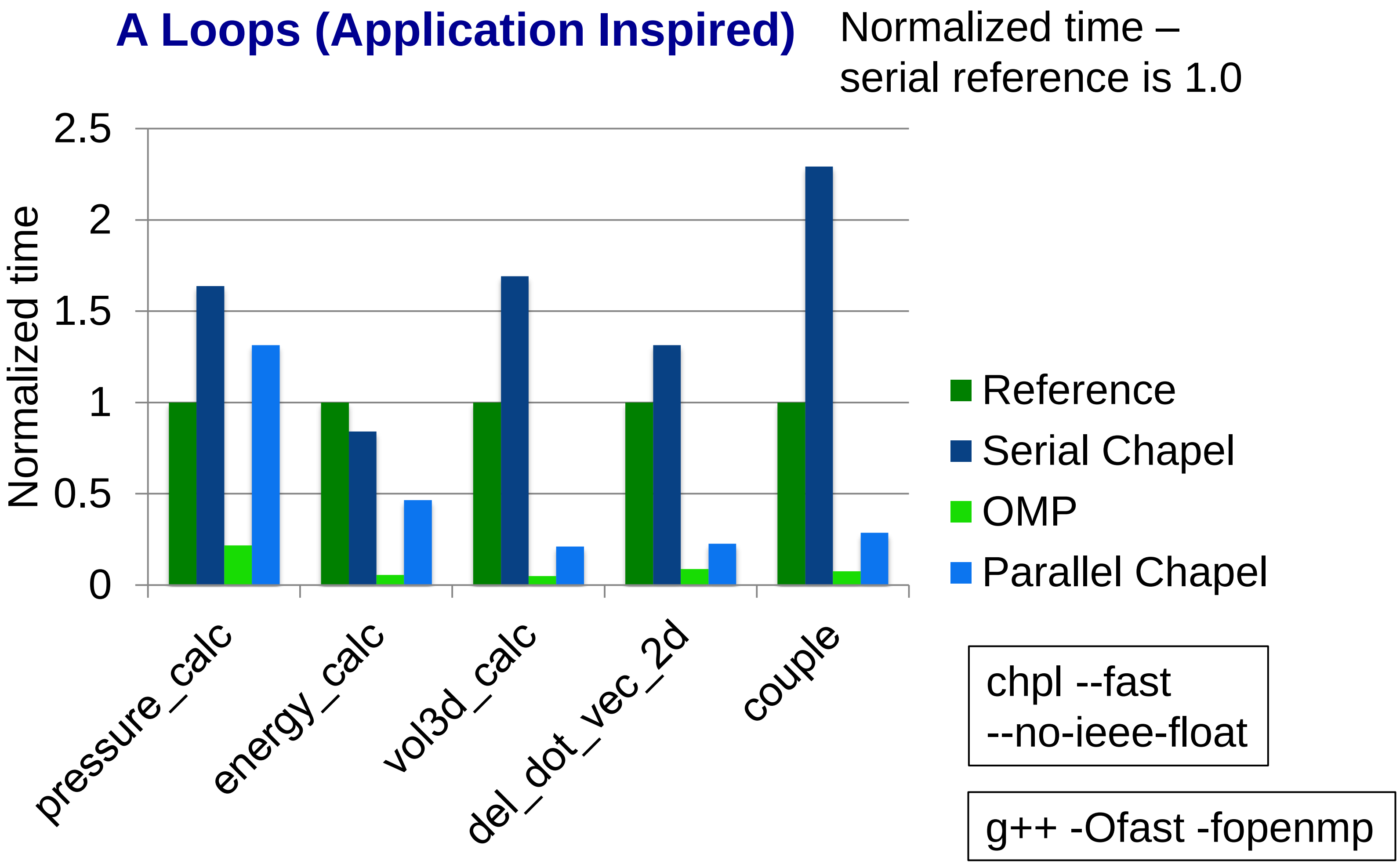
LCALS Code
Richard D. Hornung
LCALS version 1.0
LLNL-CODE-638939
2013

- **Three loop subsets (30 kernels total)**
  - Subset A: Loops representative of application codes
  - Subset B: Simple, basic loops
  - Subset C: Loops extracted from “Livermore Loops coded in C”
- **Each kernel is run for three sizes (Short, Medium, Long)**
- **Each kernel is implemented in a number of “variants”**
  - RAW (traditional C usage), OpenMP, C++ template-based, etc.

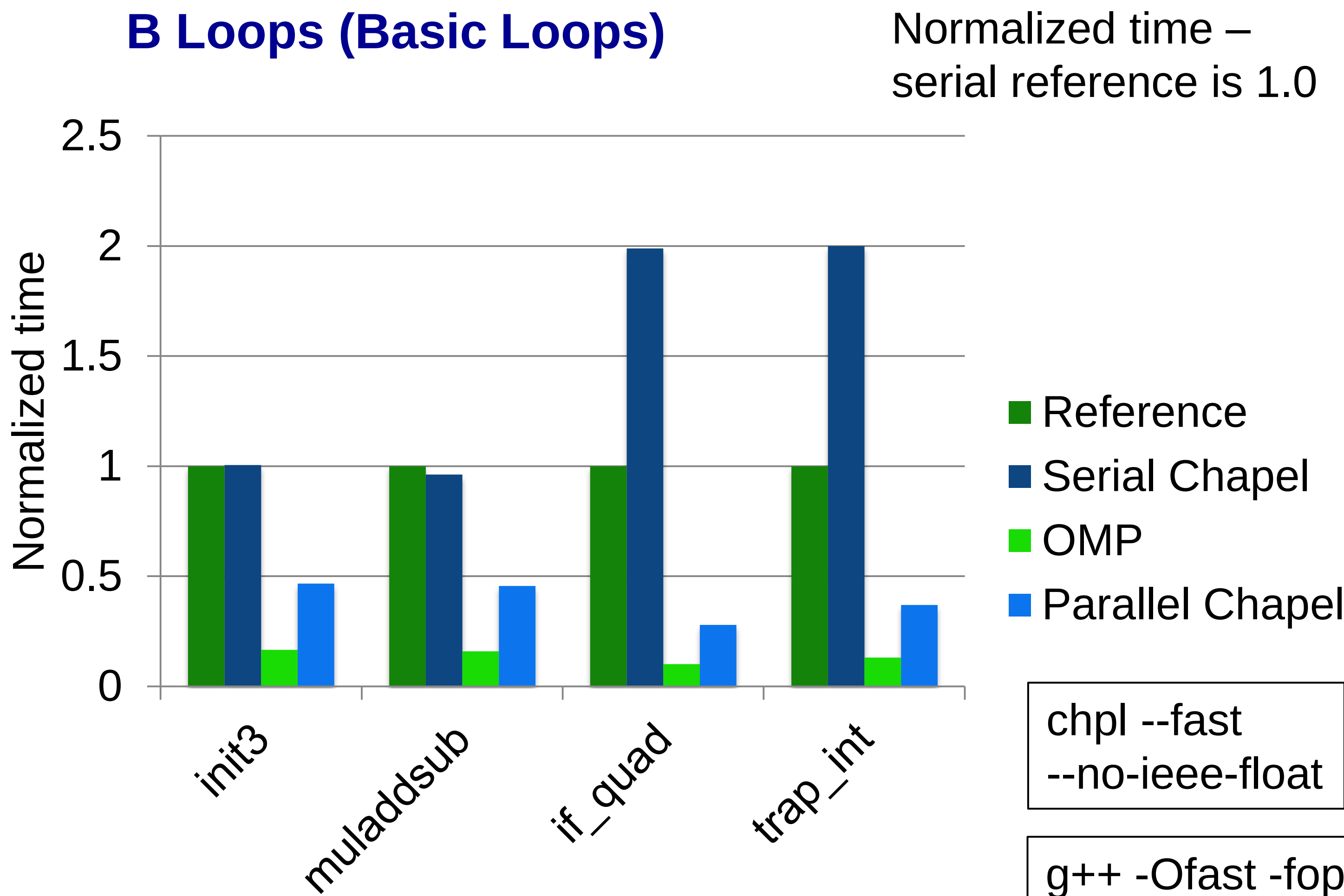
LCALS: This Effort

- **Port LCALS framework to Chapel**
  - ~2400 lines of Chapel framework code
- **Port the RAW and RAW+OpenMP kernels to Chapel**
  - RAW: All 30 kernels ported and getting correct results
  - RAW+OMP: All 11 kernels ported and correct
    - RAW+OMP kernels are a modified subset of the RAW kernels
  - ~2200 lines of Chapel kernel code
- **Compare performance vs. reference versions**
  - Executed on one Cray XC40 compute node
  - 24 Intel Xeon cores per node
  - Compiled with: gcc 5.3.0
  - The following graphs show results for the “Long” size

LCALS: Performance Comparison



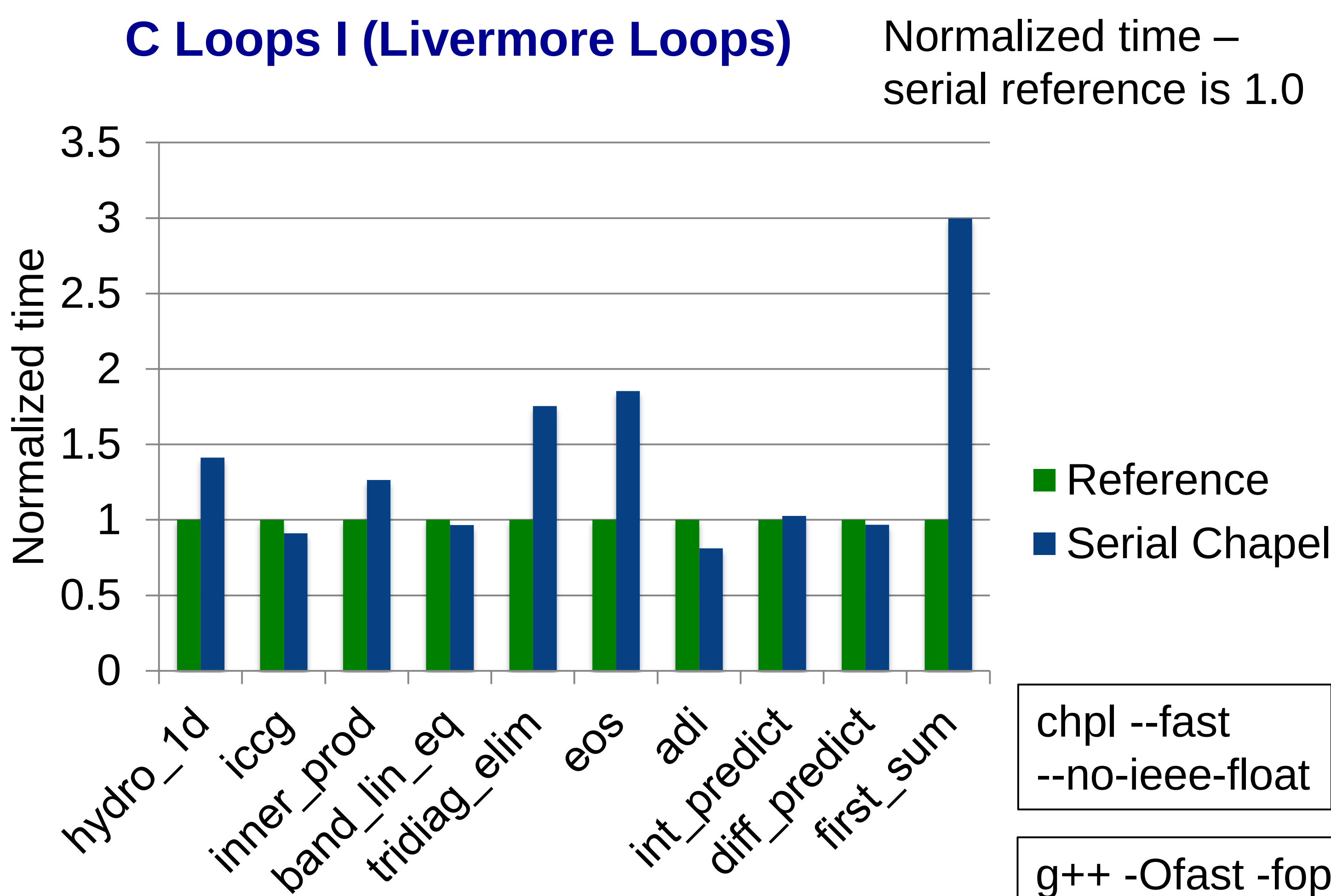
LCALS: Performance Comparison



chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp

LCALS: Performance Comparison

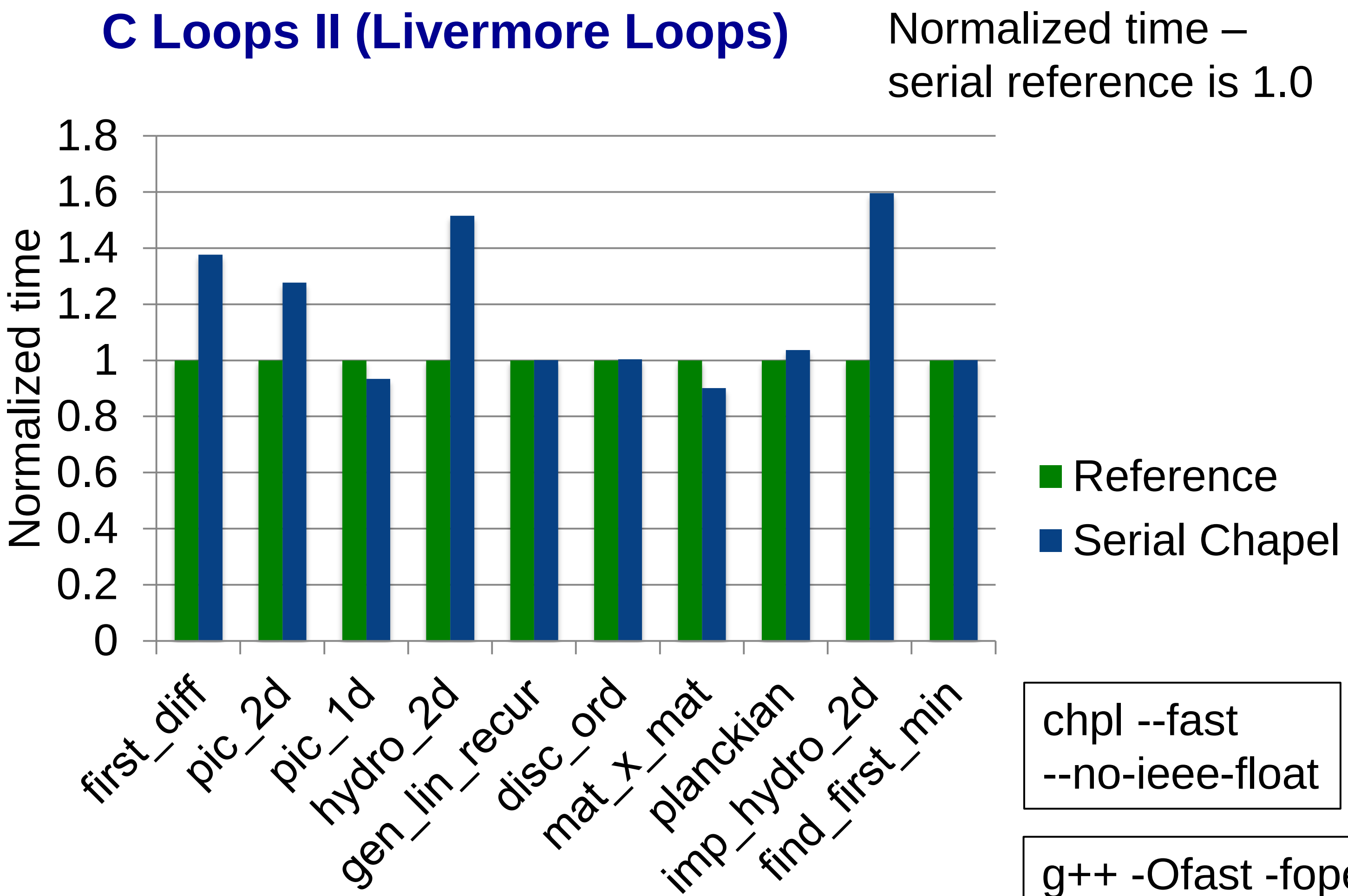


Normalized time – serial reference is 1.0

chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS: Performance Comparison



Normalized time – serial reference is 1.0

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp

LCALS: Status

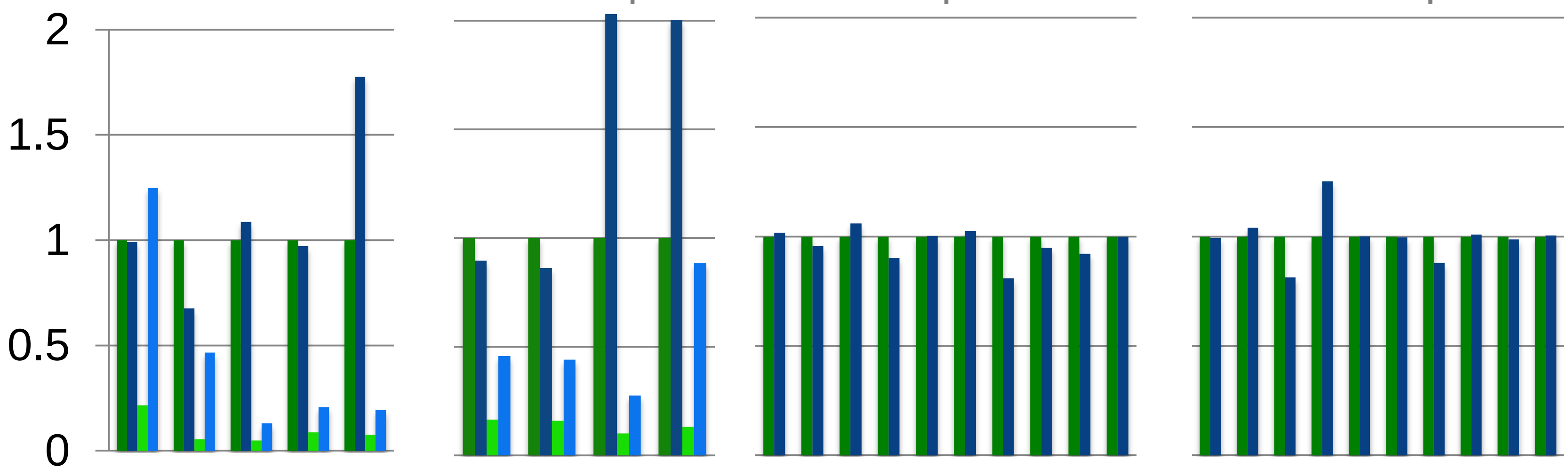
- **RAW and RAW+OMP kernels ported**
- **Performance is hit-or-miss**
  - serial performance:
    - ~half of the serial kernels are competitive with the reference versions
    - a handful are >2x off (e.g. first_sum, couple, …)
    - the remainder need some attention
  - parallel performance:
    - generally lagging significantly relative to OpenMP
- **Tracking performance of serial RAW kernels nightly**

  http://chapel.sourceforge.net/perf/chapcs/?graphs=lcalsshort,lcalsmedium,lcalslong
- **Chapel LCALS port included in the 1.13 release**

  `$CHPL_HOME/examples/benchmarks/lcals`

LCALS: Array Inner Multiplications

- **One cause of missing LCALS performance is known:**
  - Chapel uses an integer multiply for an array's innermost dimension
  - Unnecessary for typical arrays, only more advanced ones
    - e.g., rank-change, reindexing of strided slices, …
  - For typical cases, adds overhead relative to C
    - Ongoing work is striving to eliminate multiplies in these cases
  - Meanwhile, can be squashed manually using a config param
    - results in dramatic serial performance improvements for most loops:



LCALS: Next Steps

- **Eliminate inner multiplies when unnecessary**

- **Understand causes of remaining performance differences**
  - focus on serial outliers, parallel cases
  - compare vectorization of Chapel code with reference versions
  - identify other overheads in generated code

- **Get the parallel kernels into nightly performance testing**

- **Explore more elegant Chapel loop expressions**
  - Use whole-array operations, array slicing, etc.

MiniMD Benchmark Improvements

MiniMD: Background

- **MiniMD: "Mini Molecular Dynamics"**
  - Proxy application from Sandia's Mantevo group
  - Represents key idioms from real applications

- **Initially written as an intern project in 2013**
  - First major exploration of stencil codes in Chapel
  - Utilizes a custom variant of the *Block* distribution: *StencilDist*
  - Available in the release since that summer:
    $CHPL_HOME/examples/benchmarks/miniMD

- **Largely untouched since then**
  - Until now!

MiniMD: Correctness/Style Improvements

For this release…

- **Fixed bounds-checking bugs**
  - Incorrect logic in non-periodic cases
  - Incorrect bounds-checking with RAD optimization

- **Fixed the iterator that yields ghost/fluff/boundary cells**
  - Failed to correctly yield all overlapping regions on each locale

- **Switched to reduce-intents instead of atomics**
  - When first written, reduce intents did not exist
  - Using atomics is ugly and diverges from the reference

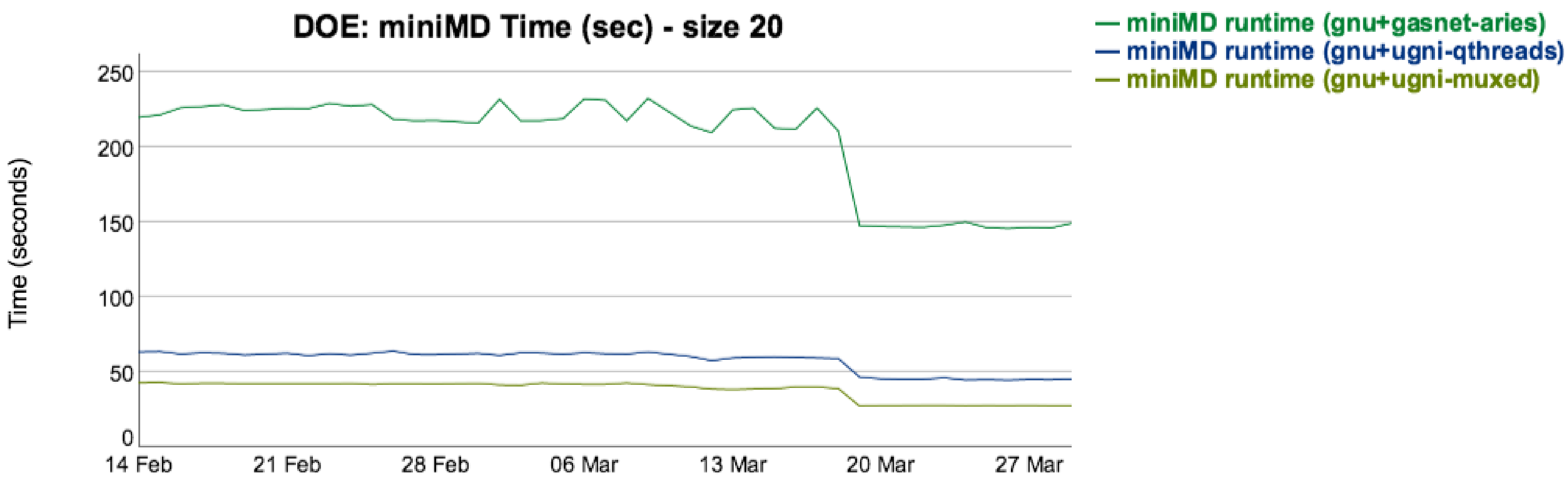
MiniMD: Performance Improvements

- **Improved parallelization for the exchange step**
  - Implemented in *StencilDist*'s updateFluff() method
  - Comprises most of the communication at scale

- **Leveraged forall-intents to reduce communication**
  - Used the 'in' intent to copy data across locales just once

MiniMD: Impact

- **Stencil distribution is better overall**
  - Fewer correctness issues
  - Special features are faster
- **~25% performance boost**
  - 16-node Cray XC results:



MiniMD: Next Steps

- **Improve Stencil distribution performance further**
  - Address known issues with array-of-arrays
  - Leverage bulk transfer optimization

- **Stencil distribution improvements**
  - Strive for more elegant ways to use this distribution
  - Explore promotion of *StencilDist* to $CHPL_HOME/modules/dists/
  - Clarify relationship between *StencilDist* and *BlockDist* (unify?)

The ISx Benchmark in Chapel

ISx: Background

- **ISx: Scalable Integer Sort benchmark**
  - modern replacement for NPB IS to address its shortcomings
  - developed at Intel, published at PGAS 2015: [paper](#) [slides](#)
  - computation style:
    - local SPMD-style computation with barriers
    - punctuated by all-to-all bucket exchange pattern

- **SHMEM and MPI reference versions available on GitHub**
  https://github.com/ParRes/ISx

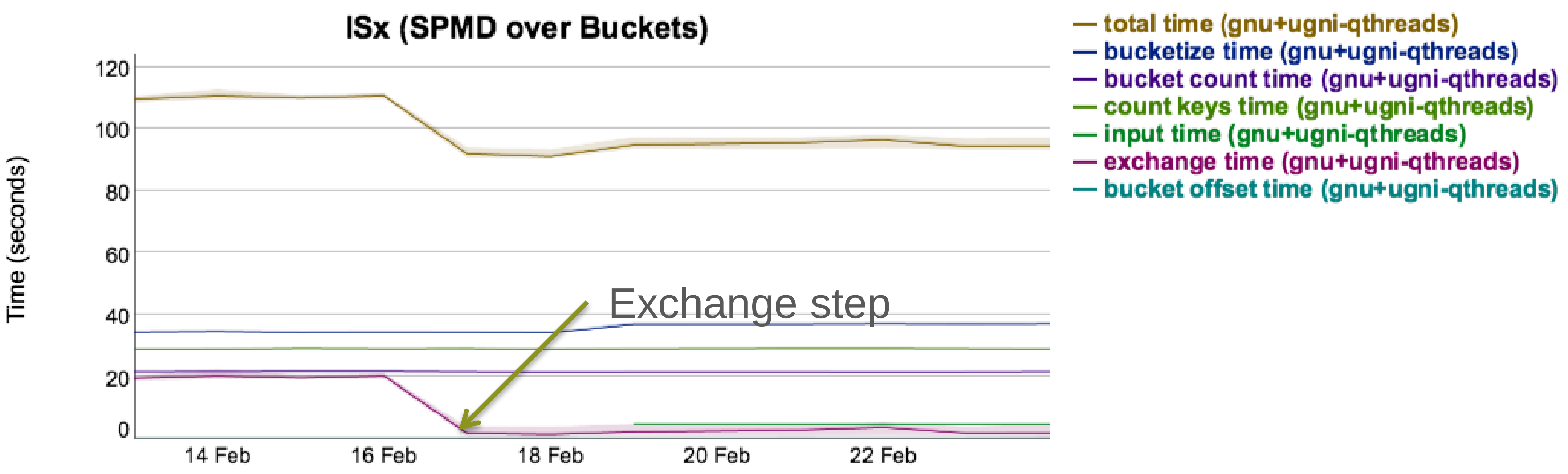
- **A good case study for Chapel**
  - a common parallel pattern for distributed memory programming

ISx: This Effort

- **Ported ISx to Chapel**
  - Initial port co-developed with Jacob Hemstead, an author of ISx
- **Developed multiple variations in different styles**
  - Pure SPMD with coforalls
    - Most similar to the SHMEM reference version
  - “Global view” with locales, atomics, and forall loops
- **Investigate performance bottlenecks**
  - All graphs shown here were gathered on 16 nodes of Cray XC

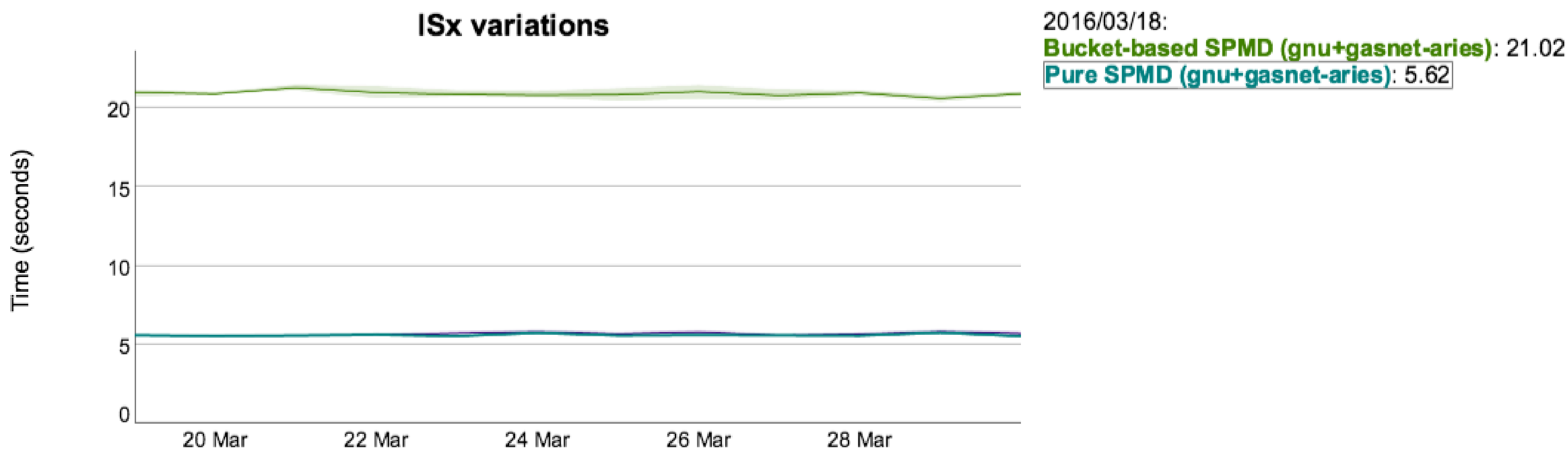
ISx: Performance – Bulk Transfer

- **At scale, the exchange step takes most of the time**
  - Expressed in Chapel via assignments between array slices
  - Ought to benefit from bulk transfer optimization
- **Bulk transfer optimization wasn't firing as expected**
  - Thwarted by overly-conservative runtime check, now fixed
  - (see upcoming Bulk Transfer slides for more detail)



ISx: Performance – Atomics

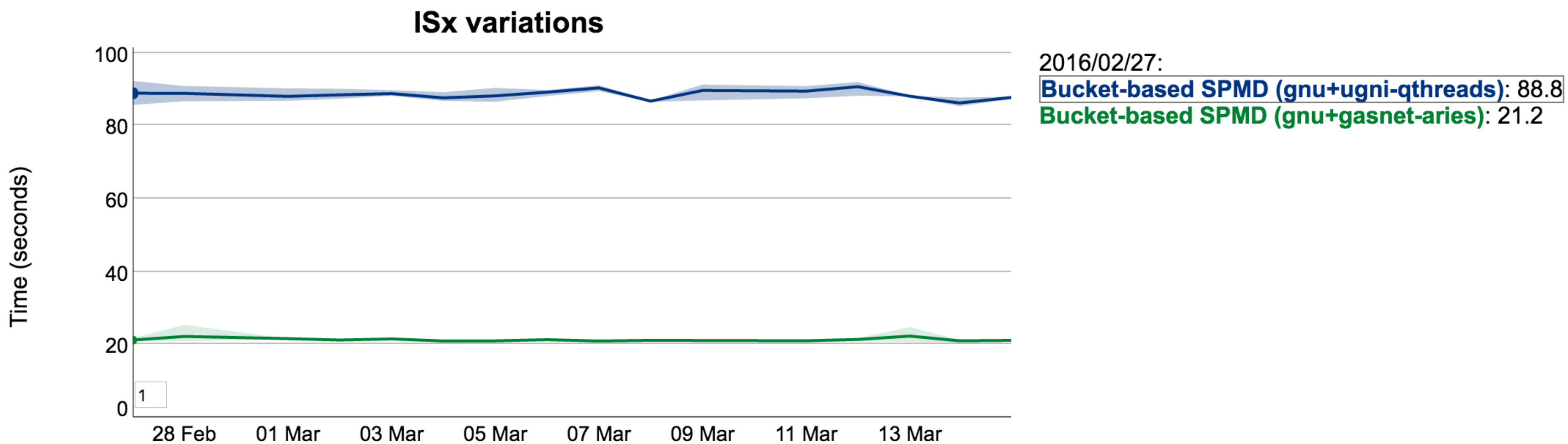
- **Global-view Chapel is slower than the SPMD variation**
  - by up to 4x!



- **Likely due to atomics**
  - Global-view uses atomics to coordinate between forall-loop iterations
  - SPMD uses serial for loops without atomics

ISx: Performance – Proc. vs. Network Atomics

- **On Crays using ‘ugni’, overhead of atomics is even worse**
  - ‘ugni’ defaults to network atomics
    - slow compared to processor atomics if only used locally, as in much of ISx
    - 4x worse than GASNet version on previous slide:



    - forcing processor atomics via CHPL_NETWORK_ATOMICS closes this gap
  - Future work:
    - automatically optimize atomics that are only used locally
    - add user-oriented capability to request local-only atomics

ISx: Performance – Loop-Invariant Expressions

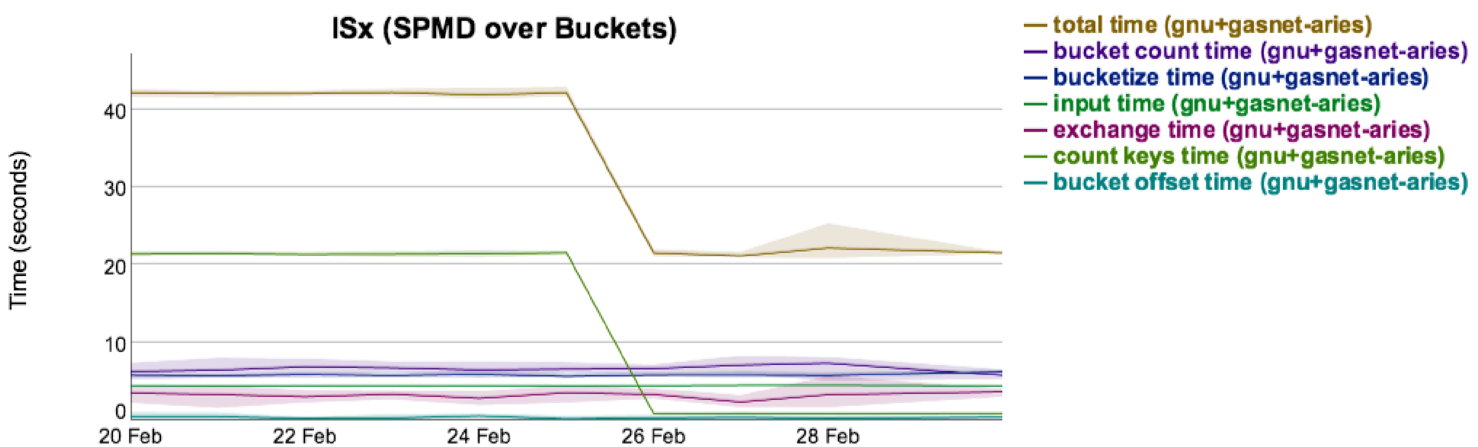
- **Manually optimized user-level code:**
  - Many loops contain loop-invariant common sub-expressions

```
for i in 0..#myBucketSize do
  myLocalKeyCounts[allBucketKeys[taskID][i]] += 1; // loop-invariant
```

  - Manually hoisting such expressions had a huge impact

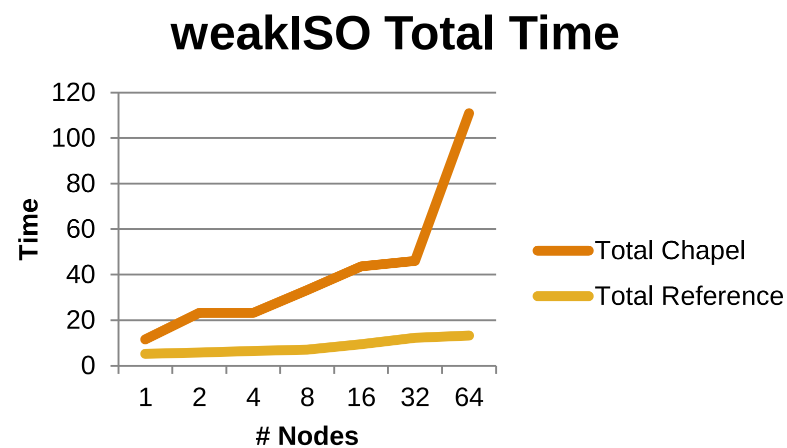
```
ref myBucket = allBucketKeys[taskID];
for i in 0..#myBucketSize do
  myLocalKeyCounts[myBucket[i]] += 1;
```

  - Compiler should automatically hoist such sub-expressions

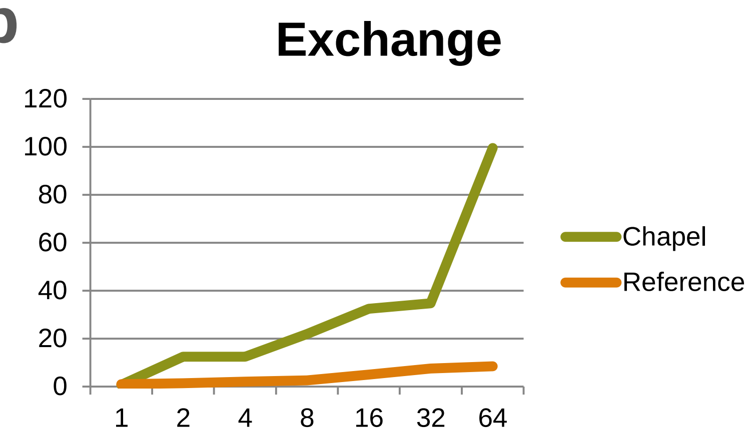


ISx: Performance Summary

- **Have made progress, but still falling off at scale:**
  - Numbers gathered on Cray XC
  - Reference version is SHMEM



- **Main bottleneck: exchange step**
  - Poor array slicing performance
  - Possible barrier performance issue
  - Plus a handful of other issues…



ISx: Status and Next Steps

Status:

- Chapel port of ISx released with 1.13
  - Just the SPMD variation for now:
    `$CHPL_HOME/examples/benchmarks/isx/`
  - Exhibits many performance and scalability issues

Next Steps:

- Continue performance evaluation and improvements:
  - Optimize slicing
  - Eliminate reference counting
  - Improve atomics and locality
  - Evaluate and optimize barriers
  - Address other miscellaneous issues
- Reduce gaps between Chapel and reference, global-view and SPMD
  - Focus on SPMD vs. reference first since computation is naturally SPMD

Bulk Transfer Improvements

Bulk Transfer: Background

- **Array assignment looks something like this:**
  - Implemented within our internal modules (simplified):

```
proc =(ref A: [], B: []) {
  if compatibleTypes(A, B) && isContiguous(A, B) then
    bulkTransfer(A, B);   // implemented with memcpy(), get(), or put()
  else
    forall (a, b) in zip(A, B) do a = b;
}
```

- **Typically, one big GET/PUT is better than many small ones**
  - bulkTransfer() above is designed to handle such cases
  - Some benchmarks stand to benefit greatly
    - e.g., ISx

Bulk Transfer: This Effort

- **Found cases where optimization didn't fire as expected**
  - Investigated and diagnosed those cases
  - Removed an incorrect / overly-conservative check in isContiguous():
    - Checked whether sliced arrays started at the beginning of the actual array:

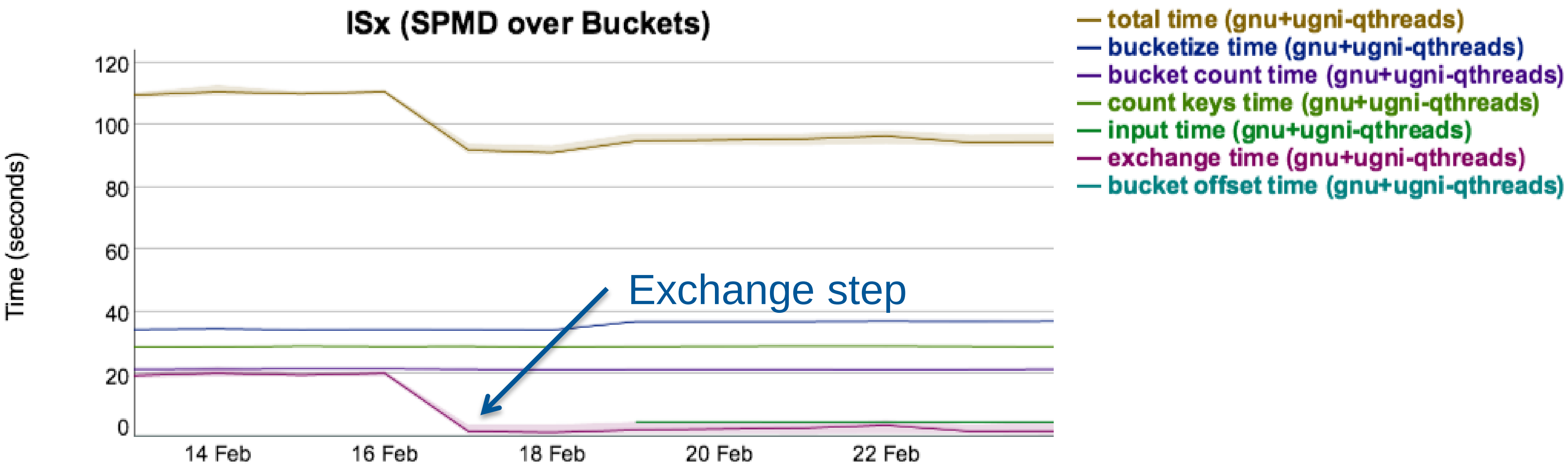
```
var A, B : [1..20] int;

// Starts at '1', successfully bulk transfers
A[1..10] = B[1..10];

// Starts halfway, bulk transfer did not fire in 1.12 (but could've!)
// Fixed in 1.13
A[10..20] = B[10..20];
```

Bulk Transfer: Impact

- **Significantly improved ISx exchange time**
  - went from ~20s to ~2s (on 16 node XC)



Bulk Transfer: Next Steps

- **Look for more opportunities to enable bulk transfer**
  - Stencil distribution?
  - Other kinds of arrays?

- **Investigate correctness/performance of strided transfer**
  - Optimization contributed by Rafael Asenjo et al. (U. Malaga)
  - Enabled by ‘useBulkTransferStride’ config param
    - Currently disabled by default due to lack of familiarity, experience, testing
    - Goal: enable by default for next release

Local On-Statements

Local-On: Background

Background:

- On-statements are used to execute code on a given locale
  - Can also target sub-locales, as in the NUMA locale model
- Compiler inserts wide pointers for references spanning on-statements
  - Not necessary for sub-locales that share memory
  - Introduces needless overhead
- Compiler can't generally distinguish sub-locales from top-level locales

Local-On: This Effort

This Effort:

- Introduce “local on” statements for sub-locales

// execute on the first child locale

```
local on here.getChild(0) {
  writeln("On sublocale ", here);
}
```

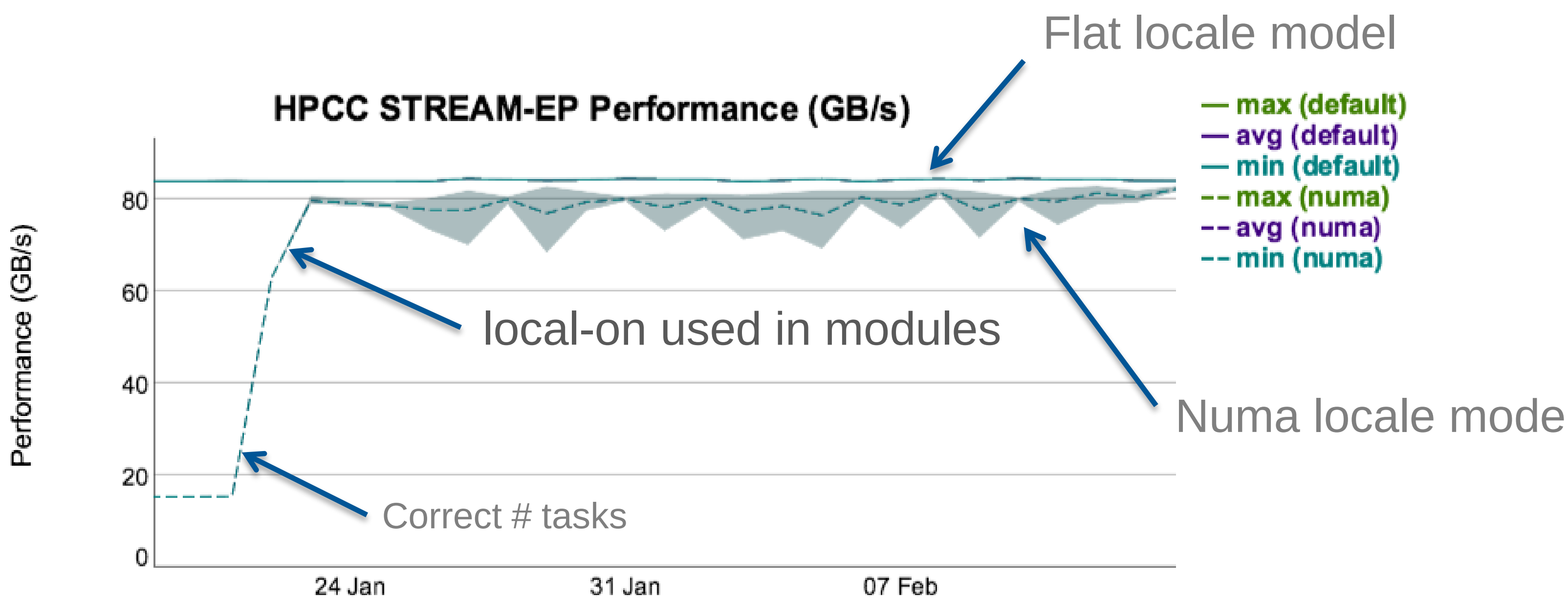
- Generate runtime error if ‘local on’ leaves current node
  - checks can be disabled by --no-local-checks, --no-checks, or –fast

// Start on Locale 0, execute on last locale

```
writeln("On Locale 0");
local on Locales[numLocales-1] { // fails here for numLocales > 1
  writeln("On locale", here);
}
```

Local-On: Impact

- **Compiler can reduce overhead for local on-statements**
- **Better performance for wide-pointer-sensitive benchmarks**
  - e.g. HPCC STREAM-EP



Local-On: Status and Next Steps

Status:

- Local-on available in 1.13
  - Used by ‘Range’ and ‘DefaultRectangular’ iterators
  - Documented online: http://chapel.cray.com/docs/latest/technotes/local.html#the-local-on-statement

Next Steps:

- Use local-on in more places as we continue NUMA work

Numa maxTaskPar Fix

Numa maxTaskPar: Background and This Effort

Background:

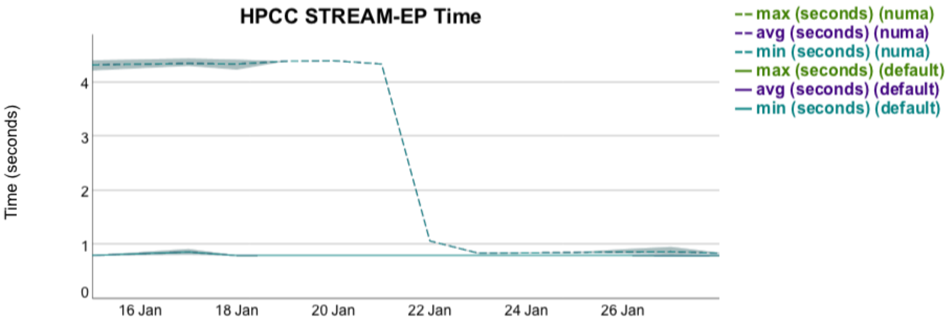
- Numa locale model set top-level *here.maxTaskPar* incorrectly
  - used *numSublocales* instead of runtime *chpl_task_maxTaskPar()*
  - e.g., for two 12-core processors, *maxTaskPar* was 2 instead of 24
- Forall loops create parallelism based on top-level *maxTaskPar*
  - so, *maxTaskPar*=2 resulted in only 2 tasks being created
- Resulted in abysmal performance for many benchmarks
  - e.g., 16 GB/s for stream using 'numa' locale model vs. 84 Gb/s with 'flat'

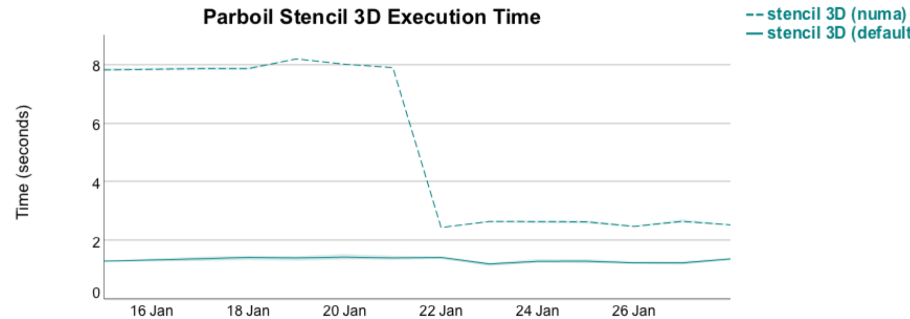
This Effort:

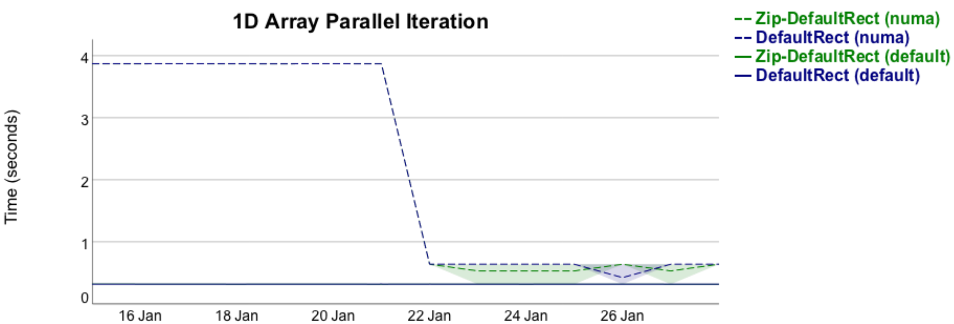
- Set top-level *here.maxTaskPar* correctly
  - improved stream performance using 'numa' locale model to 63 GB/s

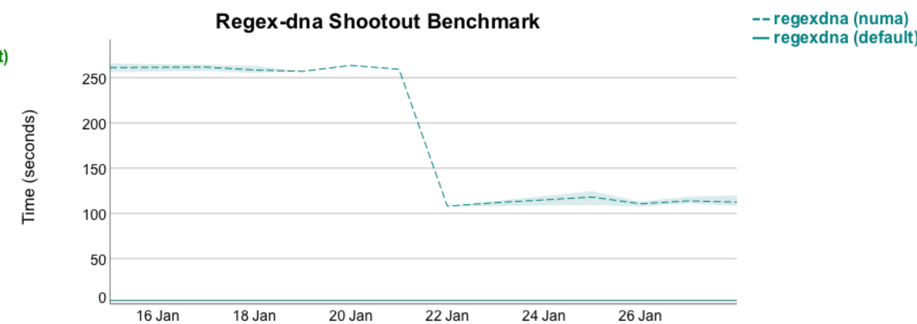
Numa maxTaskPar: Impact

- Resulted in dramatic performance improvements









Array-as-Vector Improvements

Array-as-Vector: Background and This Effort

Background:

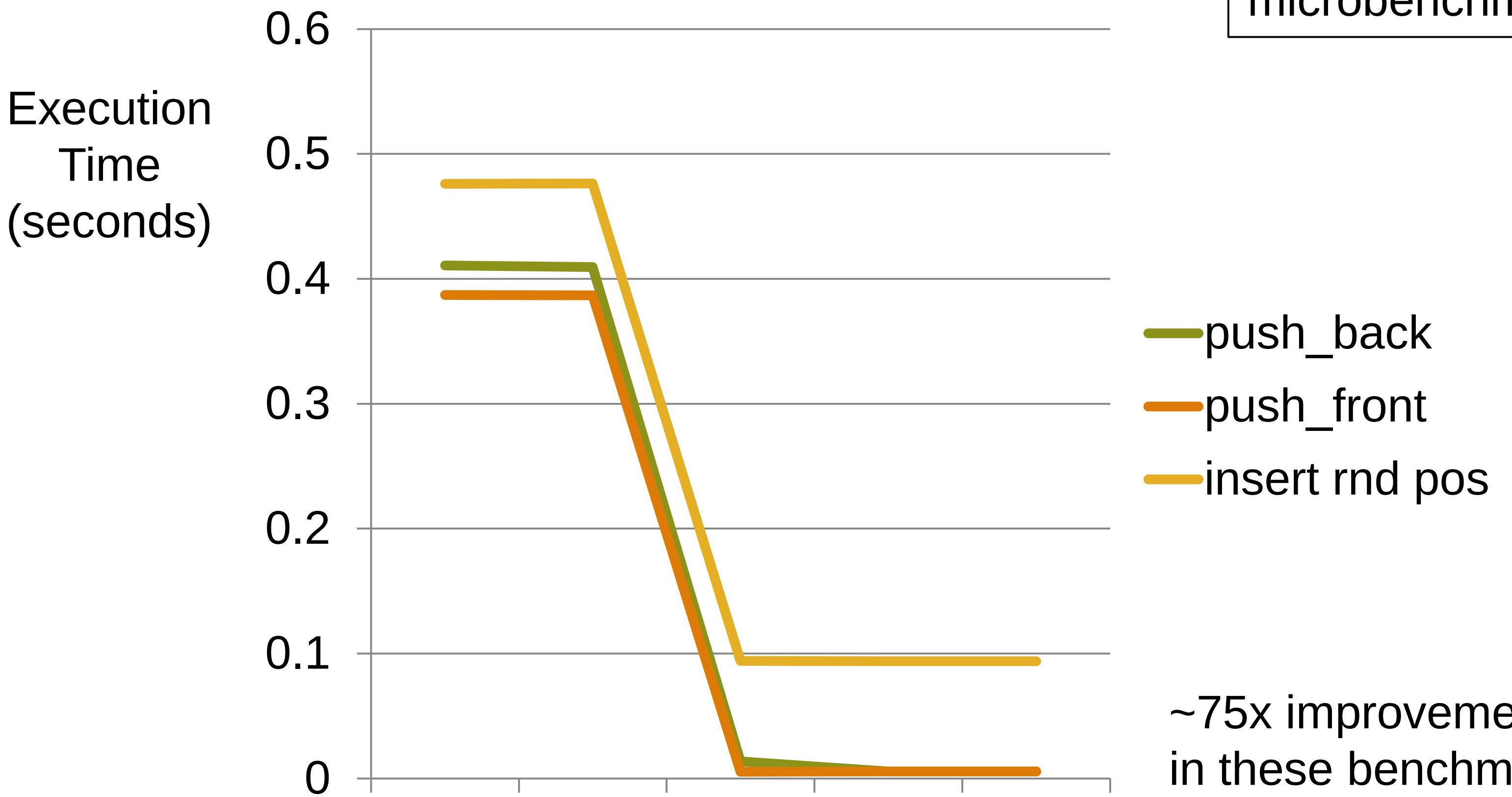
- Vector operations on arrays were added in 1.10
  - A.push_back(val)
  - A.insert(pos, val)
  - A.pop_front()
  - etc.
- Each insertion or deletion triggered an array reallocation
  - Never wasted any space
  - But *very* slow!
  - (goal at that time was to get feature up and running, tune later)

This Effort: Amortize the allocation overhead ("later" is now!)

- Grow/shrink backing array by a factor of its current size
  - Factor is adjustable using a config param (defaults to 1.5)
- Size of backing array is invisible to the end-user
  - Size tracked by a *range* field added to arrays

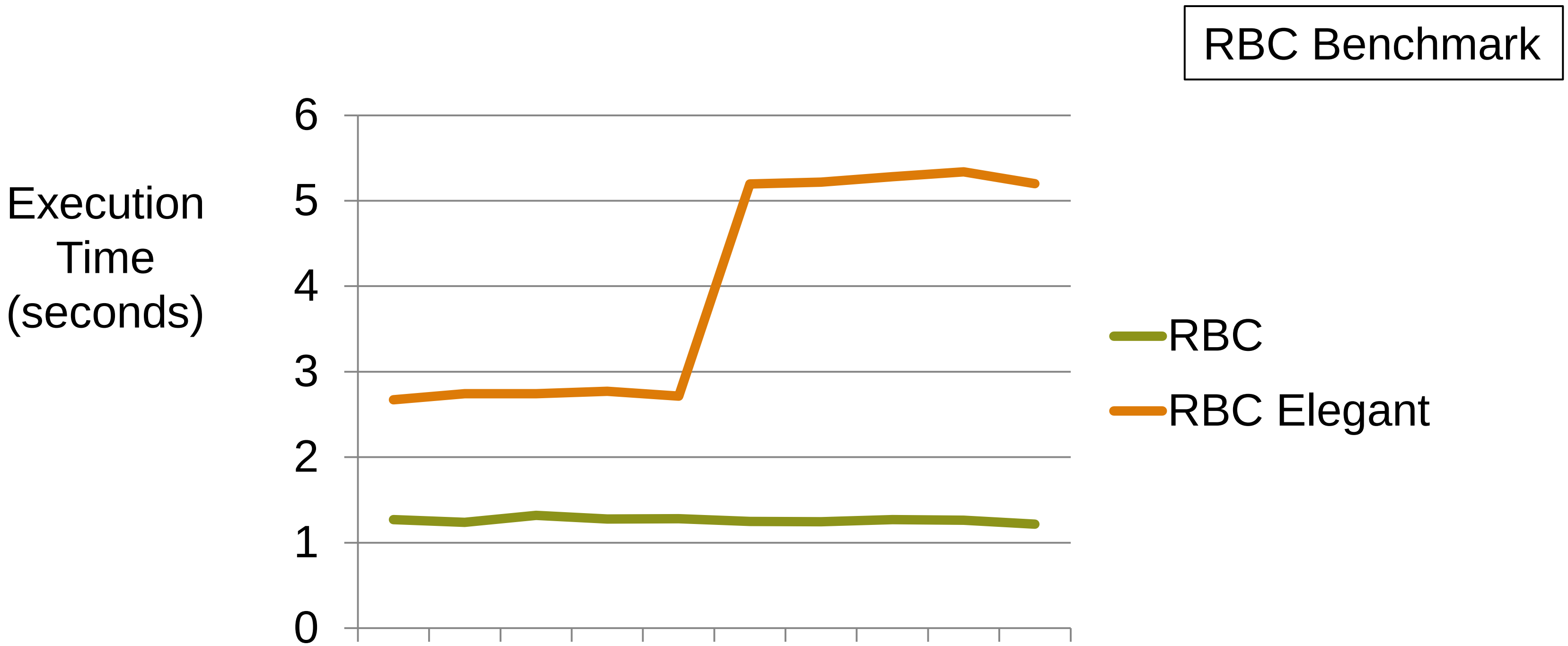
Array-as-Vector: Impact

Array-as-vec microbenchmarks



~75x improvement in these benchmarks

Array-as-Vector Improvements: Impact



- **Performance regression in RBC benchmark**
  - Yet, RBC doesn't use array-as-vector feature
  - Caused by (unused) range added to arrays to track allocated size
  - Motivates optimizing away range field when it won't be used
    - potentially using 'void' field approach mentioned in language deck

Array-as-Vector Improvements: Next Steps

- **Unify parallel safety vs. performance story across types**
  - associative/sparse domains/arrays have similar tensions
- **Make shrinking of buffer less aggressive**
  - adds overhead for unfortunate push/pop pairs at boundary sizes
- **Optimize the range field away for non-array-as-vec arrays**
  - Arrays with rank != 1
  - Arrays that are stridable
  - Analyze operations applied to given arrays, conservatively (?)
- **Tune the default growth factor**

Reduction Performance Improvements

Reduction Performance: This Effort

Background: Reduce expressions had a custom implementation

- independent of forall loop implementation in spite of similarities
  - e.g., standalone iterator was never considered
- performance lagged behind forall loops as well as a result

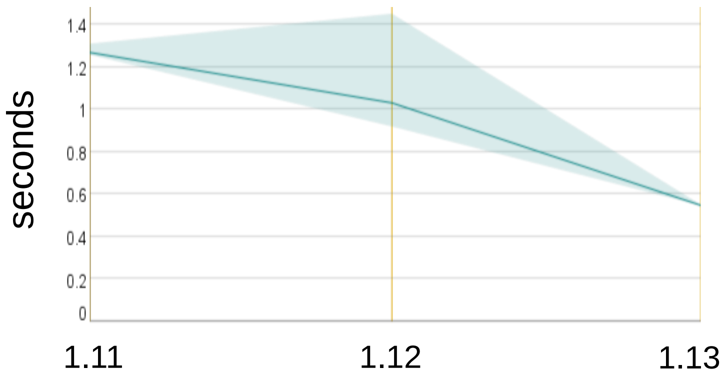
This Effort: Implement reductions using forall loops

- leverage reduce intents to perform reduction
  - only some built-in operations are handled at present
    `+, *, &, |, &&, ||, ^, min, max`
- as before, use serial loop when parallel iterator is not available

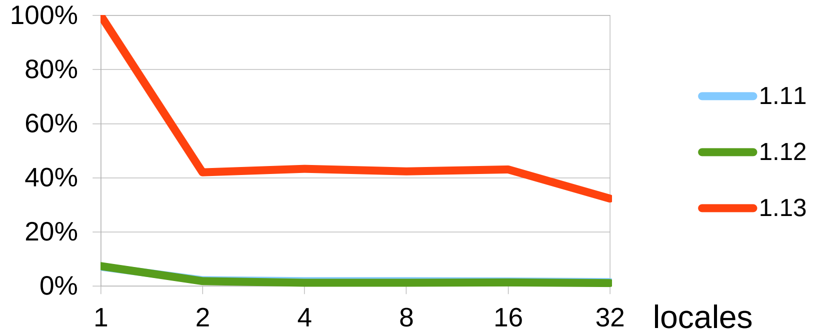
Reduction Performance: Impact – *Reduction*

Benchmark: a single plus-reduction over a large array

single-locale time



multi-locale efficiency
vs. 1.13 on 1 locale (0.307s)



- 1.12 to 1.13 speedup: 1.9x
  - 1.11 to 1.13 speedup: 2.3x
- 1.11 and 1.12 very noisy
  - reporting just best times here
- measured on Linux server

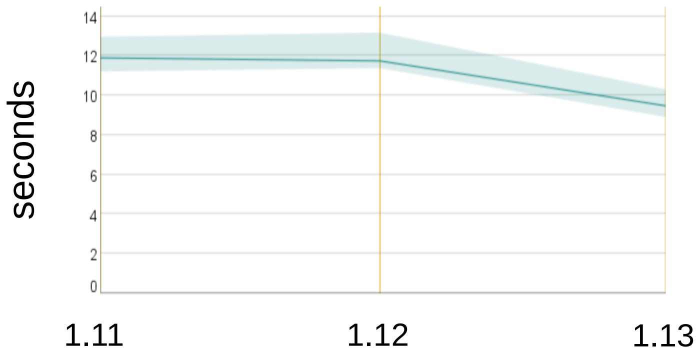
- uses weak scaling
  (constant array size per locale)
- 1.12 to 1.13 speedup: 13x to 30x
- 1.11 is comparable to 1.12
  (overlaps on graph)
- measured on Cray XC40

Reduction Performance: Impact – CG

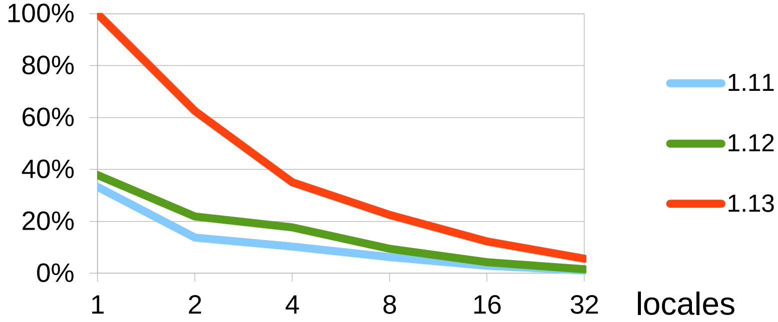
Benchmark: an iterative conjugate gradient benchmark

- earlier evaluation presented by Laura Brown at CHIUW 2015

single-locale time



multi-locale efficiency
vs. 1.13 on 1 locale (17.16s)



- 1.12 to 1.13 speedup: 1.2x
- measured on Linux server

- multiple reductions, smaller arrays
- strong scaling (constant grid size)
- 1.12 to 1.13 speedup: 2x to 3.5x
- measured on Cray XC40

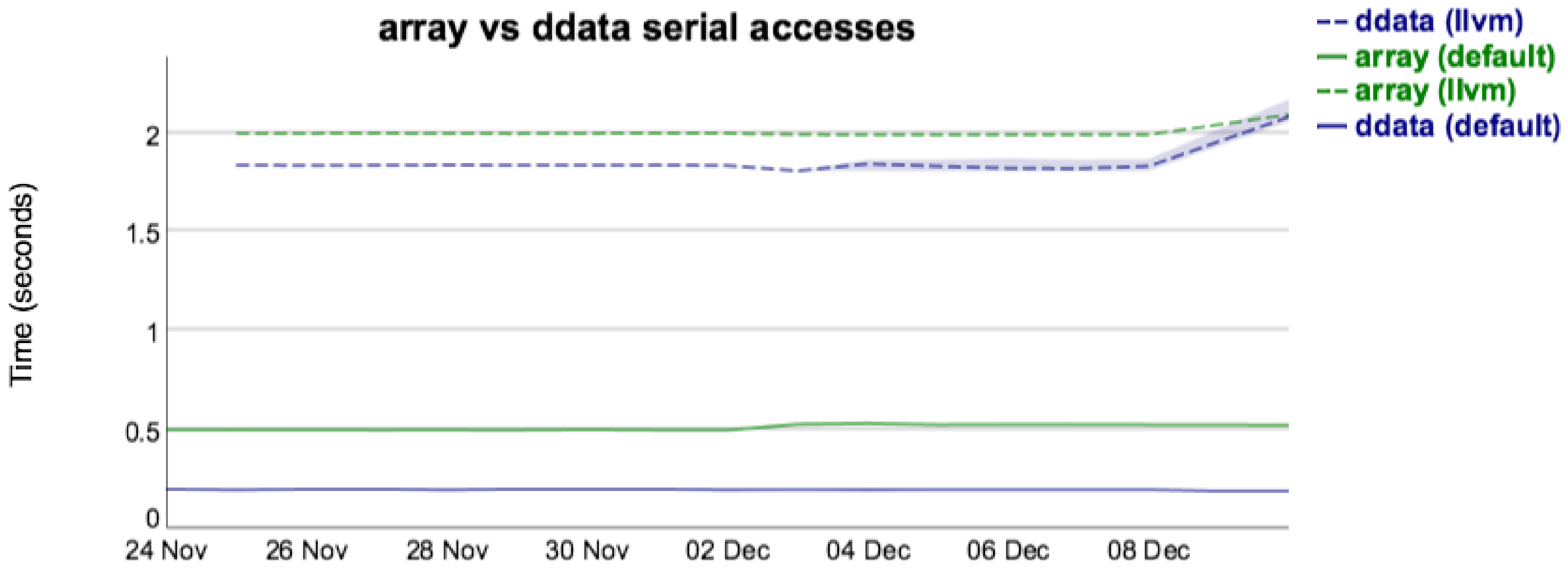
Reduction Performance: Next Steps

- **Extend these improvements to other cases**
  - arbitrary reduction operators
    - related to support for generalizing reduce intents (see “Language” slides)
  - zippered reductions
  - forall expressions
- **Tune performance of forall loops**
  - more efficient parallel iterators
    - e.g. tree-based spawning of tasks across locales
  - lower overhead specific to reduce intents

Performance of LLVM Back-End

LLVM Performance: Background

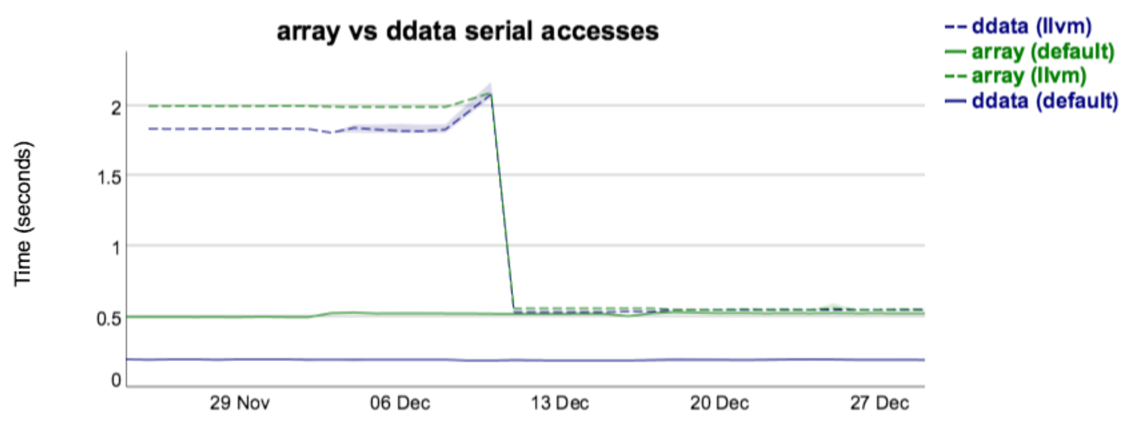
- **Noticed significant performance gaps with --llvm back-end**
  - especially for microbenchmarks
- **e.g., array performance test was about 9x slower**



- **Started regular perf. testing to help investigate the issue**

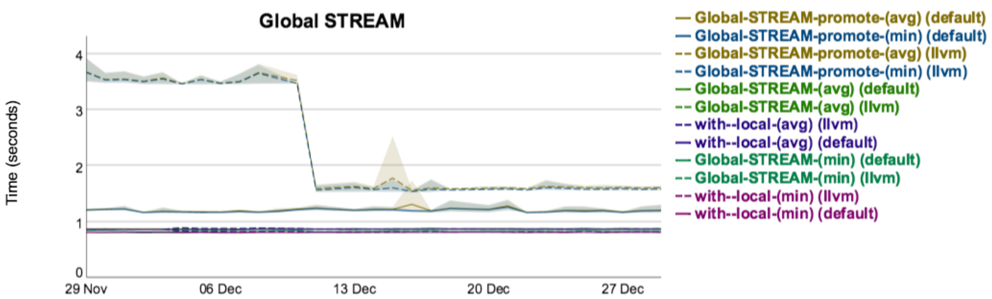
LLVM Performance: This Effort

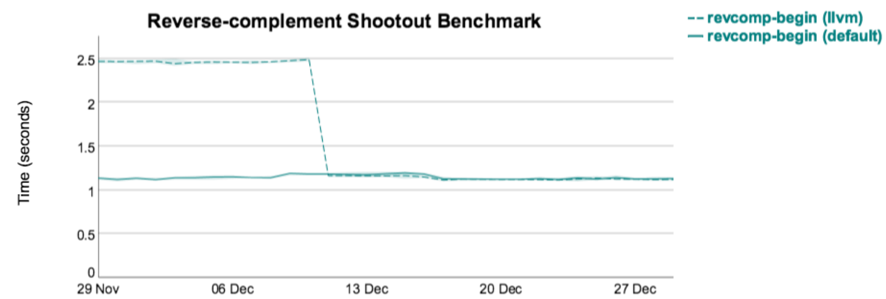
- With --llvm, *link* step actually does target code generation
- Link step was missing optimization flags
- Simply adding these flags addressed the performance gap

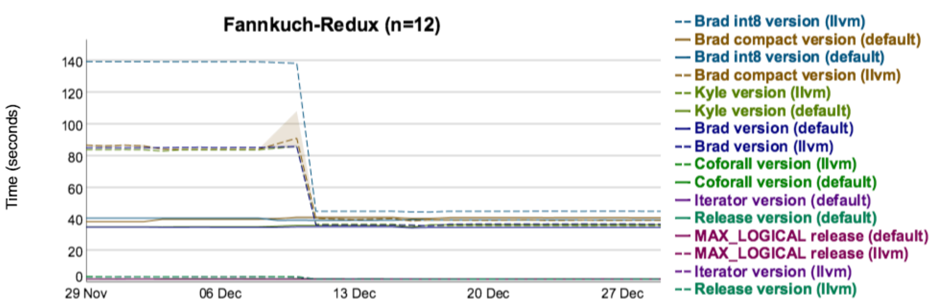


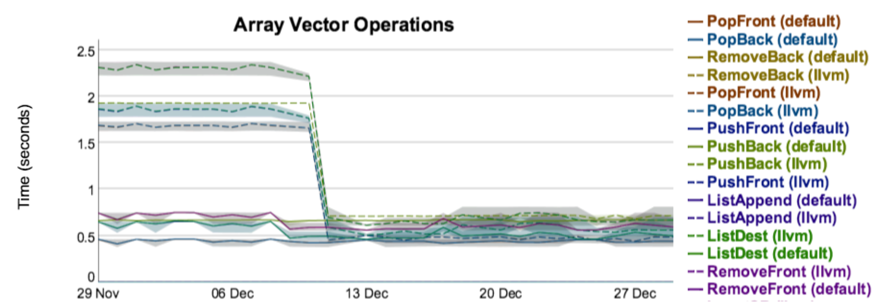
LLVM Performance: Impact

- Saw significant performance improvements



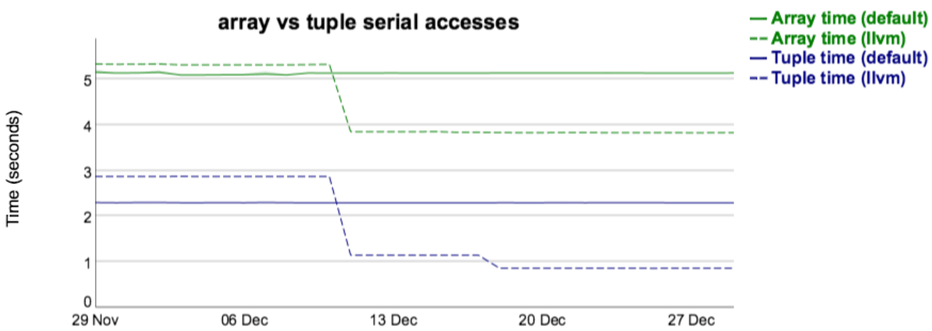


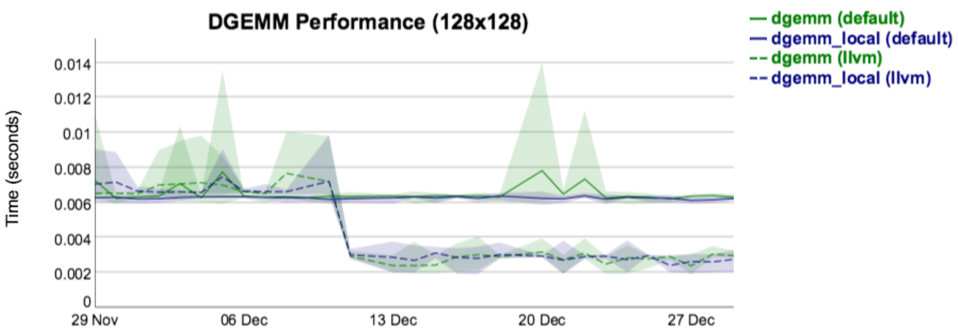


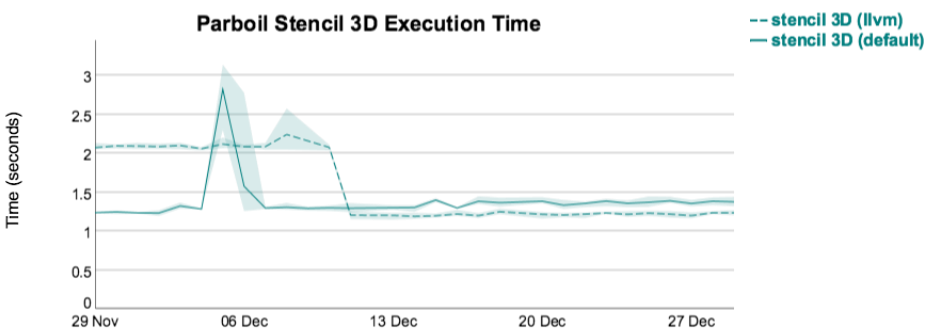


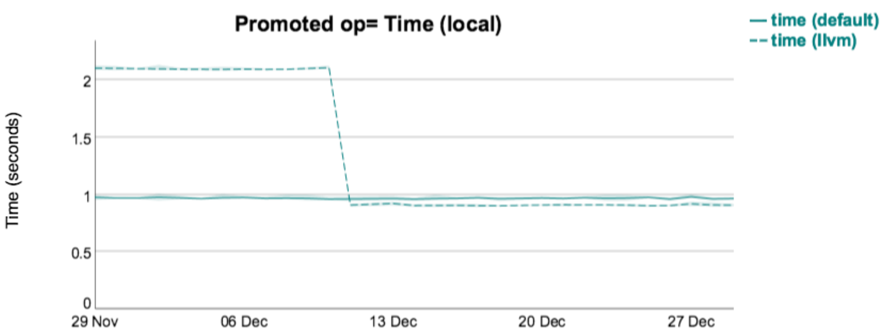
LLVM Performance: Impact

- In some cases, --llvm now outperforms the C back-end









LLVM Performance: Next Steps

Status:

- LLVM and C back-ends are now generally competitive

Next Steps:

- Revisit --llvm-wide-opt for LLVM-based multi-locale optimization:
  - Start regular performance testing
  - Change from packed wide pointers to struct wide pointers
- Improve the LLVM IR that Chapel generates:
  - add loop vectorization hints in LLVM IR for forall/vectorizeOnly loops
  - Improve type-based alias analysis
  - Indicate when a load is to a const variable
  - Investigate enabling the Polly polyhedral optimizer

Anonymous Counted Range Optimization

Counted Ranges: Background and This Effort

Background:

- Counted ranges are convenient for iterating a certain number of times
  - e.g. to loop 10 times just do:

```
for i in 0..#10   /*instead of*/ for i in 0..10-1
for i in lo..#10 /*instead of*/ for i in lo..lo+10-1
```

- Previous releases added an anonymous range optimization
  - only optimized simple fully-bounded ranges

```
for i in 0..9 do
for i in 0..10-1 do
for i in 0..10 by 2 do
```

  - did not optimize low-bounded counted ranges

```
for i in 0..#numIters do // common in internal and user code
```

This Effort:

- Optimize low-bounded counted anonymous ranges

Counted Ranges: Impact

- **Eliminated construction of low-bounded counted ranges**

```
for i in 0..#10 do writeln(i);
```

previous generated code:

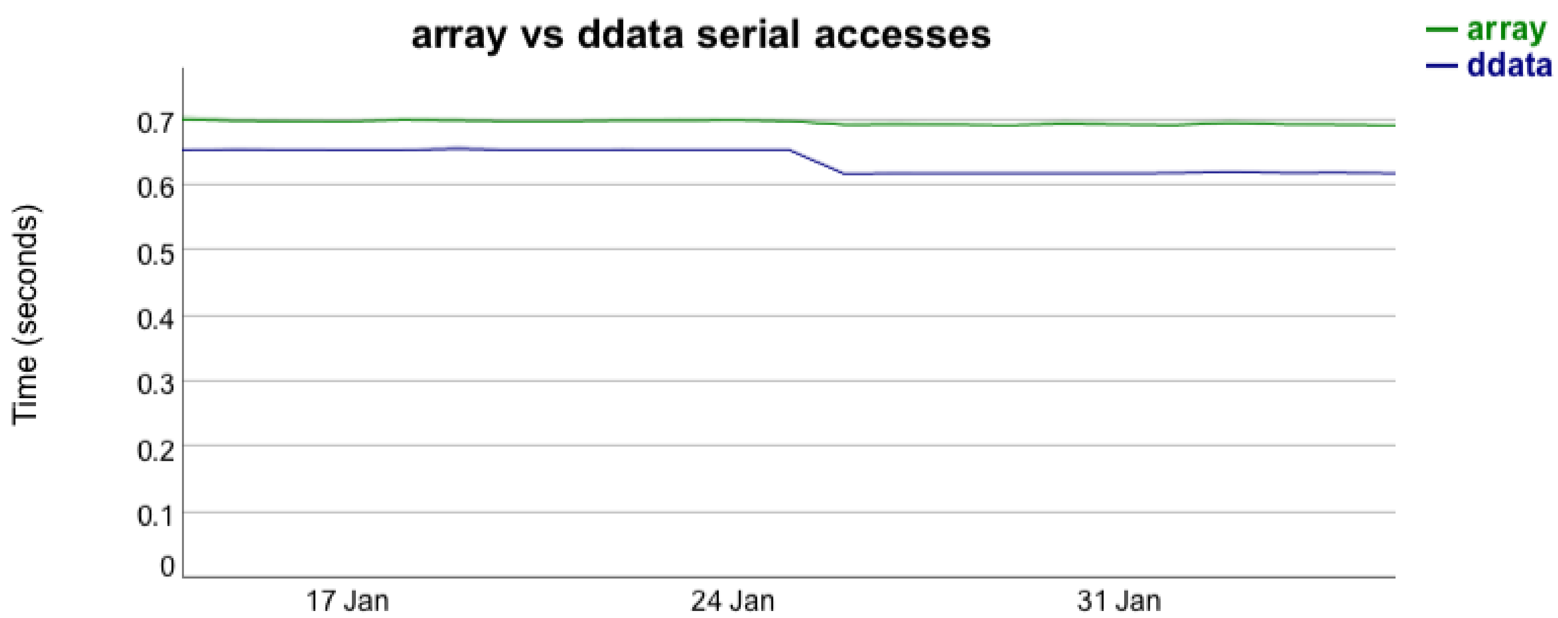
```
chpl_build_low_bounded_range(INT64(0), &call_tmp_l); // simple constructor
chpl___POUND(&call_tmp_l, INT64(10), &call_tmp_b);   // ~10 branches
low = (&call_tmp_b)->_low;
end = (&call_tmp_b)->_high;
for (i = low; i <= end; i += INT64(1))
  writeln(i);
```

now:

```
for (i = INT64(0); i <= INT64(9); i += INT64(1))
  writeln(i);
```

Anonymous Range Opt: Impact (continued)

- **Minor speedup for test that happens to use nested ranges**



- **Nice generated code cleanup**
  - but no major performance impact on most of our benchmarks

Lexical Scoping Improvements

Lexical Improvements: Background

Background:

- Chapel used to keep variables alive past lexical scopes
- For version 1.12, we decided to stop doing this
  - Reported on in the v1.12 release notes and language evolution docs
- However, many cases were still using the old semantics
  - Certain logical stack variables were allocated on the heap
    - Caused memory leaks in cases that were not reference-counted
    - Resulted in generated code complexity as well as overhead

```
{
  var x: int;                 // Surprisingly, 'x' would be heap-allocated, just
  begin with (ref x) {        // in case this 'begin' outlived x's lexical scope.
    …x…
  }
}                             // Worse, it'd be leaked here because we'd never
                              // implemented reference counting for such cases.
```

Lexical Improvements: This Effort

This Effort:

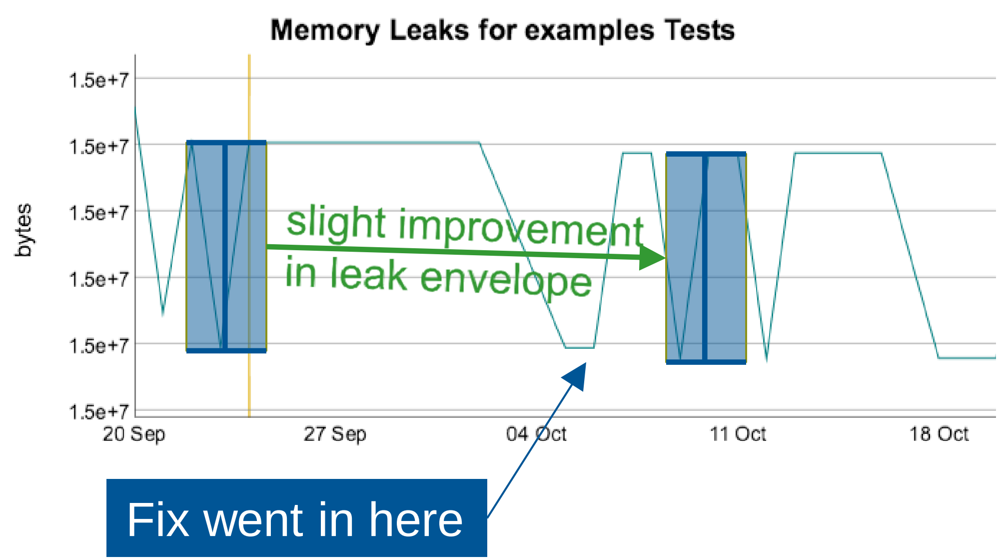
- Tighten up such cases
  - Stop allocating stack variables on the heap due to ‘begin’ statements

```
{
  var x: int;                  // ‘x’ is now stack allocated.
  begin with (ref x) {         // A reference to ‘x’ is now taken for the ‘begin’.
    …x…
  }
}                              // Like any other stack variable, ‘x’ is freed here.
                               // Any task still referring to ‘x’ is a user error
```

Lexical Improvements: Impact

- **Eliminated all heap-converted data leaks in nightly testing**
- **Minor impact in our total overall leaks**
  - Not surprising given how rarely such cases come up…
    - ‘begin’ is not used extensively in our test system
    - Even when it is, stack variables typically use default intent (const copy-in)
      - i.e., previous task intent work already closed most such leaks

```
diff --git a/memleaks.2015-10-02.nightly.log b/memleaks.2015
index 0fde8a2..fc2a0db 100644
--- a/memleaks.2015-10-02.nightly.log
+++ b/memleaks.2015-10-07.nightly.log
@@ -276,7 +276,6 @@ Number of leaked allocations
              1            24  UntypedField(complex(128))
              1            24  TypeAliasField(real(64))
              1            24  GenericClass(int(64))
-             2            16  local heap-converted data
              1            16  List(int(64))
              1            16  LocBlock(4,int(64))
              1            16  LocCyclic(2,int(64))
@@ -291,7 +290,7 @@ Total leaked memory (bytes)
 ==========================================================
```



Lexical Improvements: Other Impacts

- **Moves Chapel one step closer to being leak-free**
  - Users whose codes rely on begins will be much happier

- **Revealed a ‘ugni’ network atomic limitation with stack vars**
  - Now fixed and included in the Cray module for 1.13

Lexical Improvements: Status and Next Steps

Status:

- Chapel now implements its lexical scoping rules better
- A historical source of memory leaks is now plugged

Next Steps:

- Implement checks to help protect users from ref-after-free issues
- Close remaining leaks, particularly arrays and domains
  - these have traditionally been reference counted, though not very well
    - and in distributed cases have been intentionally leaked in spite of ref counting
  - new semantics ought to reduce or eliminate need for reference counting

Evaluation of Current Memory Leaks

Memory Leaks: Background

- **Memory leak statistics are collected every night**
  - Performance team reviews every week
  - Currently gathering single locale leaks only
- **Two metrics are tracked:**
  1. Total bytes leaked
     - Subject to test parameters (e.g., choice of array sizes)
  2. Number of tests with leaks
     - Some tests run in multiple variations, so one oversight leads to many leaks

Tests run	4,817
Tests with leaks	1,073
Total memory allocated (MiB*)	31,614
Total memory leaked (MiB)	965

April 11, 2016
* 1 KiB = 1024 bytes

Memory Leaks: Leaks by Byte

- **Small number of tests dominate overall bytes leaked**
  - Primarily due to distributed arrays

Application	Tests	Leaked (MiB)	Fraction %	Total %
studies/hpcc/PTRANS	1	805.1	81.5	81.5
optimizations/bulkcomm	8	55.7	5.6	87.1
studies/hpcc/HPL	3	39.2	4.0	91.1
users/aroonshama	8	27.9	2.8	93.9
studies/amr	2	18.7	1.9	95.8
benchmarks/ssca2	5	13.9	1.4	97.2
lammps/shemmy	1	6.9	0.7	97.9
studies/ssca2	5	6.1	0.7	98.6
benchmarks/miniMD	1	5.0	0.5	99.1

Memory Leaks: Distributed Arrays

- **Another case of suffering from old lexical scoping rules**
  - Traditionally, feared freeing distributed arrays prematurely
    - “what if some asynchronous task somewhere is still referring to it?”
  - Unfortunately, we simply chose not to free them “for now”
    - another case of “we’ll get to that later” (and again, “later is now!”)
    - codes that use a fixed number of global distributed arrays get away with it
      - many simple benchmarks are like this
    - but clearly not acceptable in general / real applications

- **As in preceding cases, new semantics help us greatly**
  - yet work remains to leverage them
  - this is arguably our top priority for 1.14

Memory Leaks: Leaks by Test Count

Source	Count	%
User fails to reclaim memory	~400	37.3
Distributed arrays	~190	17.7
Sync/single	~155	14.4
Tuples of records	~100	9.3
Initialization of generic fields	~80	7.5
Field initializer	~40	3.7
First-class functions	~25	2.3
main(args : [] string)	~20	1.9
Runtime types	~15	1.4
Misc and further classification required	~50	4.7
Total	**1,073**	

Memory Leaks: User fails to reclaim (37.3%)

- **Many simple tests leak classes/buffers in trivial ways**

```
class C {
  var x : int;
}

var c = new C(1);

writeln(c);
```

// No delete for c, so it leaks

- Leak-freedom was not the original point of the test
- Generally easy to resolve
  - In a few cases, removing the leak could change the intent of the test

Memory Leaks: Distributed arrays (17.7%)

- **Driven by original language semantics**
  - Arrays could outlive lexical scopes due to asynchronous tasks
- **Semantics now revised, but implementation work required**
  - Continue to refine record implementation
  - Improvements to constructor/destructor semantics

Memory Leaks: Sync/Single (14.4%)

- **Driven by original language semantics**
  - Syncs/singles could outlive lexical scopes due to asynchronous tasks
    - similar to array case
  - Semantics now revised, but implementation has not been updated
- **Syncs/singles currently implemented as a class**
  - However, user is not expected to delete syncs/singles
  - We intend to revise the implementation
    - Likely by conversion to a record-wrapped class
- **Most sync/single leaks are due to use of 'Random' module**
  - Each instance of a *RandomStream* stores a synchronization variable
  - More than 120 of the 155 sync leaks are due to this module
  - Intend to modify the internal implementation to delete for now
    - revert once sync is leak-free

Memory Leaks: Tuples of records (9.3%)

```
const (dist, ) = (new dmap(new DefaultDist()), );
```

- **Leaks tuple components when at the module level**
  - Missing an autoDestroy during tuple construction
  - Reference count is off-by-one on the distribution
- **Does not leak components when at the procedure level**
  - Required autoDestroy is included
- **Nearly 100 tests share a single module with this pattern**

Memory Leaks: Initialize Generic Field (7.5%)

- **All uses of a sparse subdomain leak**

```
var D1  : domain(1) = { 1 .. 10 };
var DS1 : sparse subdomain(D1);
```

  - *D1* is captured by *DS1* and is not released correctly
  - *DefaultSparseDom* has a generic field

```
class DefaultSparseDom : BaseSparseDom {
  param rank : int;
  type  idxType;
  var   parentDom;
  …
```

    - An error in the compiler-generated default constructor
    - The *parentDom* field is “initialized” from *D1* twice
    - Reference count is off-by-one
  - Any class/record with this pattern will leak

Memory Leaks: Field Initializers (3.7%)

- **An array temp is created for this field initializer:**

```
class foo {
  var i : [1 .. 5] int = [ 1, 2, 3, 4, 5 ];
}
```

  - A low-level operator disables the call to autoDestroy
    - Leaks any object that requires memory management
  - Leaks for the field initialization in classes and records

- **The following does not leak:**

```
class foo {
  var i : [ 1 .. 5] int;

  proc foo() { i = [ 1, 2, 3, 4, 5 ]; }
}
```

Memory Leaks: Runtime types (1.4%)

- **Some types must be explicitly represented at run-time:**

```
const Ndom : domain(1) = { 1 .. 10 };
type  Ntype            = [Ndom] int;
```

- *Ntype* stores a reference to *Ndom*
  - Reference count for *Ndom* is incremented while initializing *Ntype*
  - No call to free *Ntype* at end of scope
  - Reference count for *Ndom* is not decremented
- *Ndom* is leaked

Memory Leaks: Status and Next Steps

Status: Remaining leaks driven by a few well-defined modes

- Distributed arrays are a large and known problem
  - Dominates metric for leaks by bytes
  - Revision to language semantics a key to resolving this
- Many tests inattentive to memory management
  - Dominates metric for count of tests that leak
  - Fixing these cases is housecleaning
- A modest number of other patterns

Next Steps: Drive leak metrics to zero for 1.14 release

- Would signal increasing maturity of the implementation
- Extend testing framework to highlight leak regressions
  - Avoid new compiler-based errors
  - Ensure new tests are clean (when practical)

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