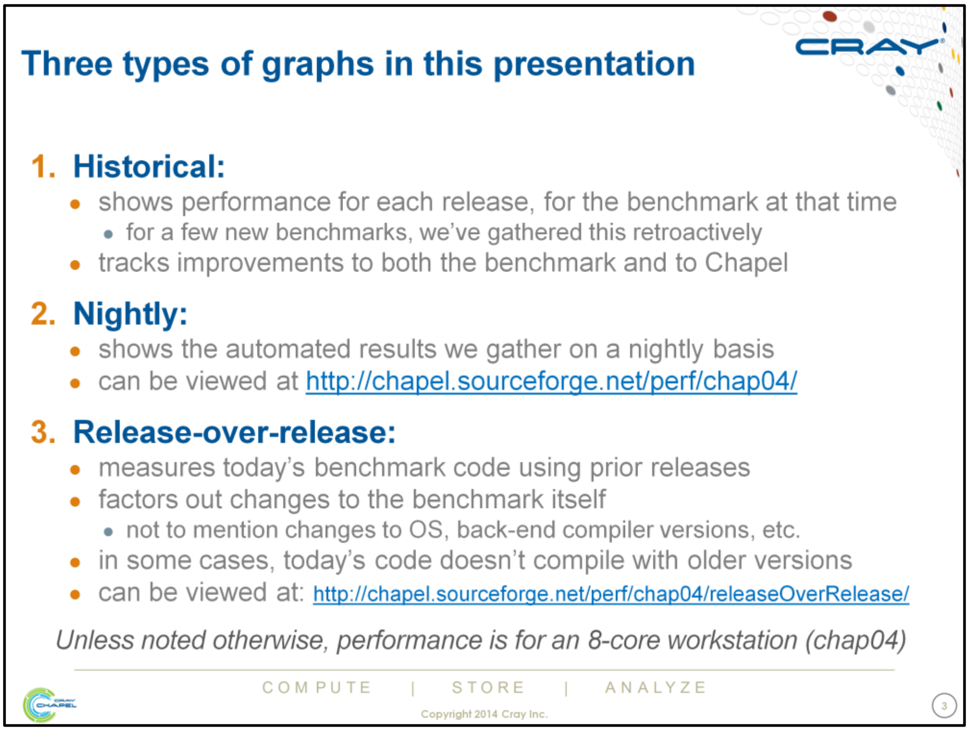
# Benchmark Improvements

Chapel Team, Cray Inc.
Chapel version 1.9
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.




Note that the vast majority of graphs in this slide deck are reporting execution times, so lower is better. The primary outliers are the HPC Challenge benchmark results near the end, which typically use performance metrics like GB/s. These cases are called out in their notes sections.

# Outline

- **Computer Language Benchmark Game ("shootout") Codes**
  - New benchmarks
    - Meteor
    - Fannkuch-redux
  - Revised benchmarks
    - Spectral-norm
    - Mandelbrot
    - Chameneos-redux
    - Fasta
  - Performance changes in stable benchmarks
    - Pidigits
    - Thread-ring
    - N-body
    - Binary-trees
- **Other Notable Single-Locale Benchmark Results**

# Computer Language Benchmarks Game

## Contest's Goal:

- see how languages compete over 13 modestly sized benchmarks
  - **performance:** serial or multicore, timing complete program execution
  - **code size:** as measured by the code's compressed size
- for more information: http://benchmarksgame.alioth.debian.org/

## Our Goal:

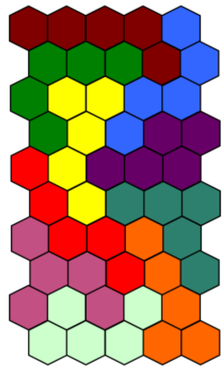
- assemble an entry that is competitive with C in performance
  - use this to raise awareness of Chapel in mainstream/open-source
- use this as a forcing function for looking at serial performance
  - the effort has had a positive impact on multi-locale cases as well

# Meteor

COMPUTE | STORE | ANALYZE

## Meteor

### Overview:
- searches for every solution to a shape packing puzzle
- unlike most shootout benchmarks, any algorithm may be used



COMPUTE | STORE | ANALYZE





Each of the 10 pieces is a unique shape that can be rotated and flipped before being placed on the board. Every piece must be used in the solution.

## Meteor

- **We have two primary versions of meteor:**

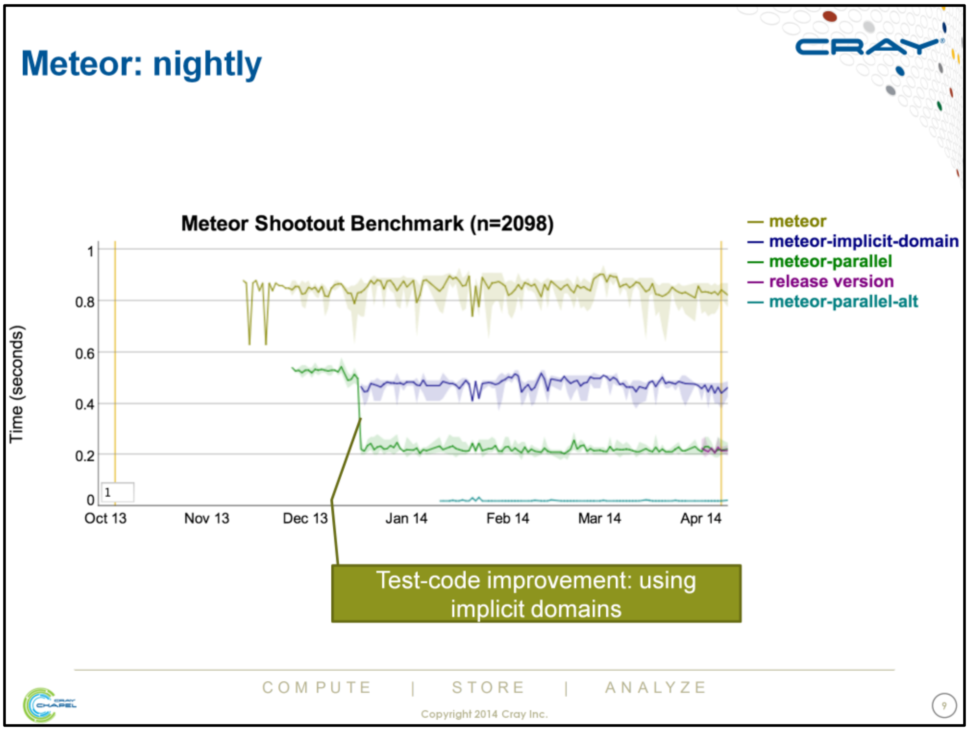
  1. meteor
     - Fairly competitive performance
     - Easy to understand
     - In the 1.9 release

  2. meteor-fast
     - Very good performance
     - Outperforms the reference version in some configurations
     - Not entirely portable due to reliance on compiler intrinsics
     - Fairly incomprehensible from an algorithm standpoint
     - Will be in the next release





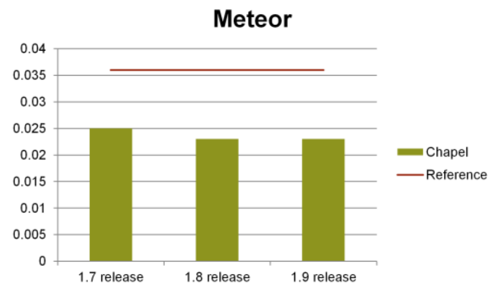
These versions are known as meteor-parallel and meteor-parallel-alt inside of the test/studies/shootout/meteor/kbrady/ directory.

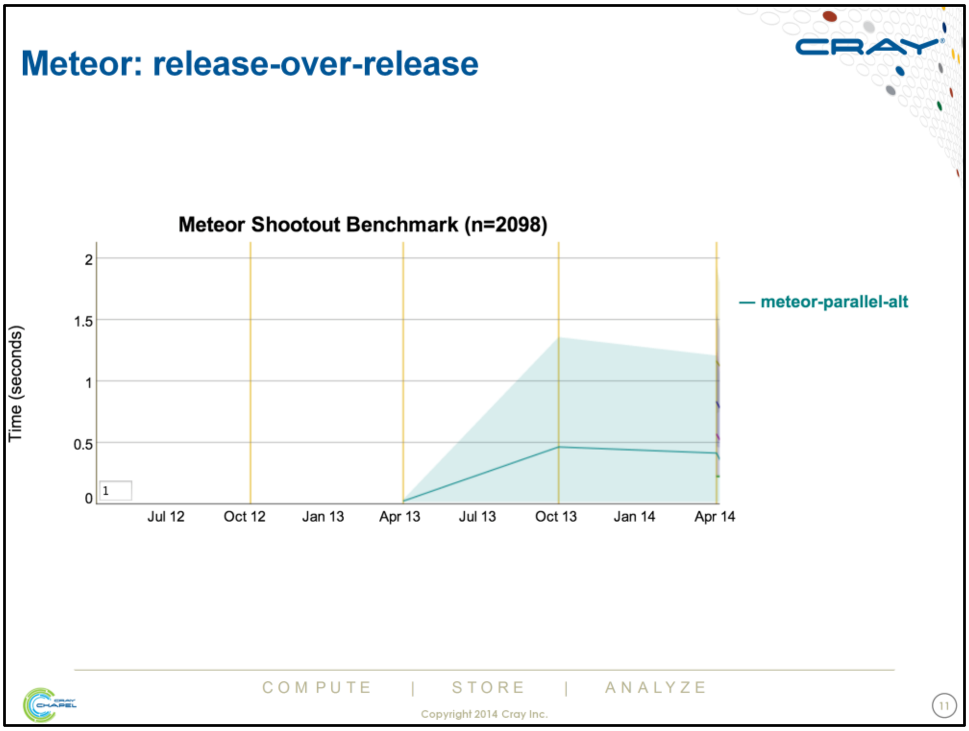



On this graph, the previously mentioned meteor is marked as 'release/meteor-parallel' and meteor-fast is 'meteor-parallel-alt'.

The series meteor and meteor-implicit-domain are single threaded versions of meteor-parallel. The gap between them is caused by a large number of copies that occur when array type is fully specified for formal arguments.

# Meteor: historical (gathered retroactively)






The noise in this graph is puzzling. For the Oct 13 and Apr 14 data points, the first of the three trials got an unexplainable timing of 1.x seconds compared to the norm which is a tiny fraction of a second (as seen in the Apr 13 timings and the nightly graph shown earlier. We're including this graph here for completeness rather than because it's particularly useful. The historical graph of the previous slide is essentially the same information (since it's all retroactively gathered) and far more indicative of the performance we typically see. It seems likely that there is some artifact in our testing system that is causing the overhead for the first run of these release-over-release timings.

## Meteor

### Next steps:

- **meteor-fast**
  - Its intrinsics can be made portable using the new BitOps module
  - Promote into the release
- **Remove penalty for fully-specified formal arguments**

```
var A: [1..n] real;

proc foo(A: [1..n] real) { … }
proc bar(A: [] real) { … }
proc baz(A: [?D] real) { … }
```

  - Of these three routines, the first is more expensive by far
    - the reason: this syntax reindexes the actual to match the formal's domain
    - the common case of the domains being the same is not optimized
  - The language, compiler, and/or modules should be updated to remove this penalty

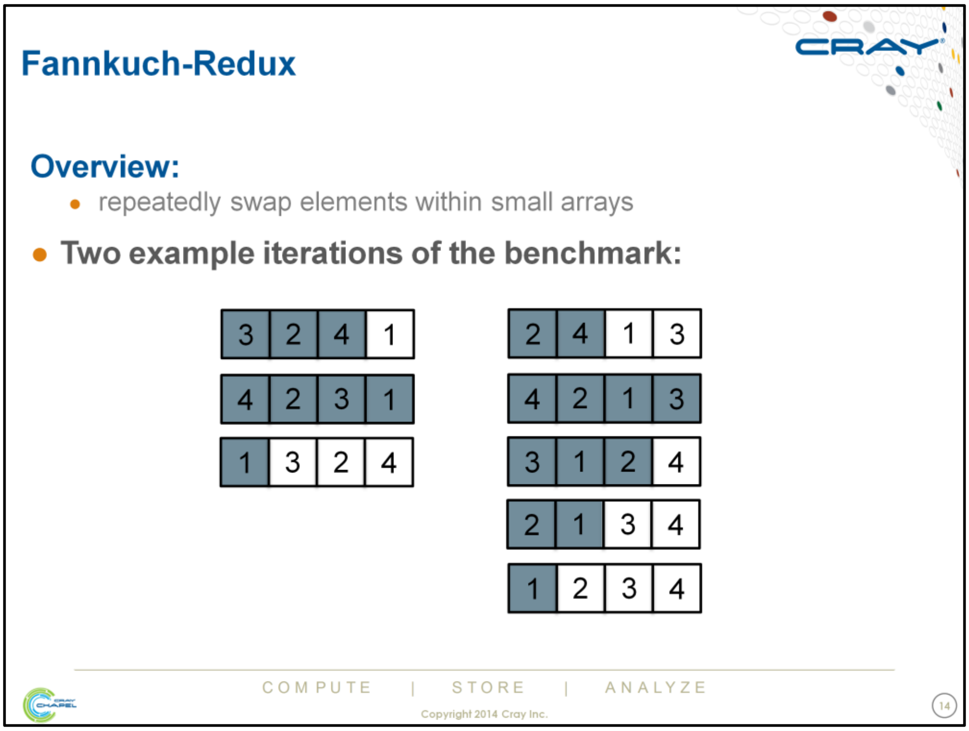






Meteor-fast was very close to being in the 1.9 release, but was held back due to portability issues. The BitOps module committed after the 1.9 release will let us fix that easily.

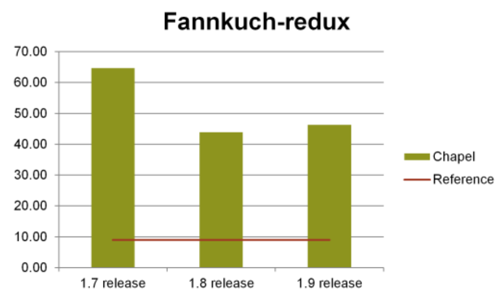
# Fannkuch-Redux



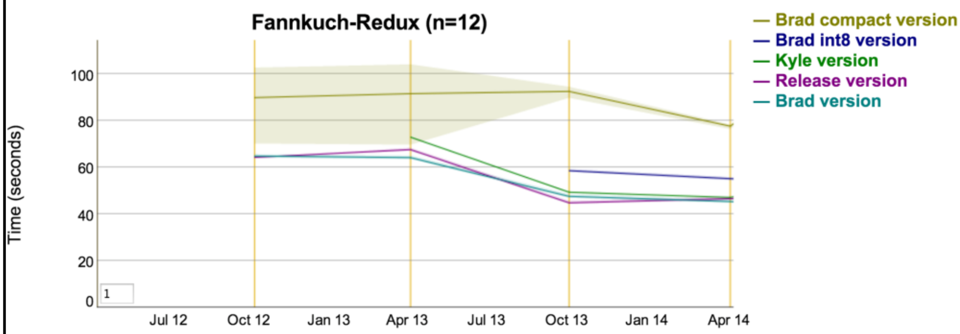

The benchmark takes every permutation of {1,…,n} and performs a few steps over them:

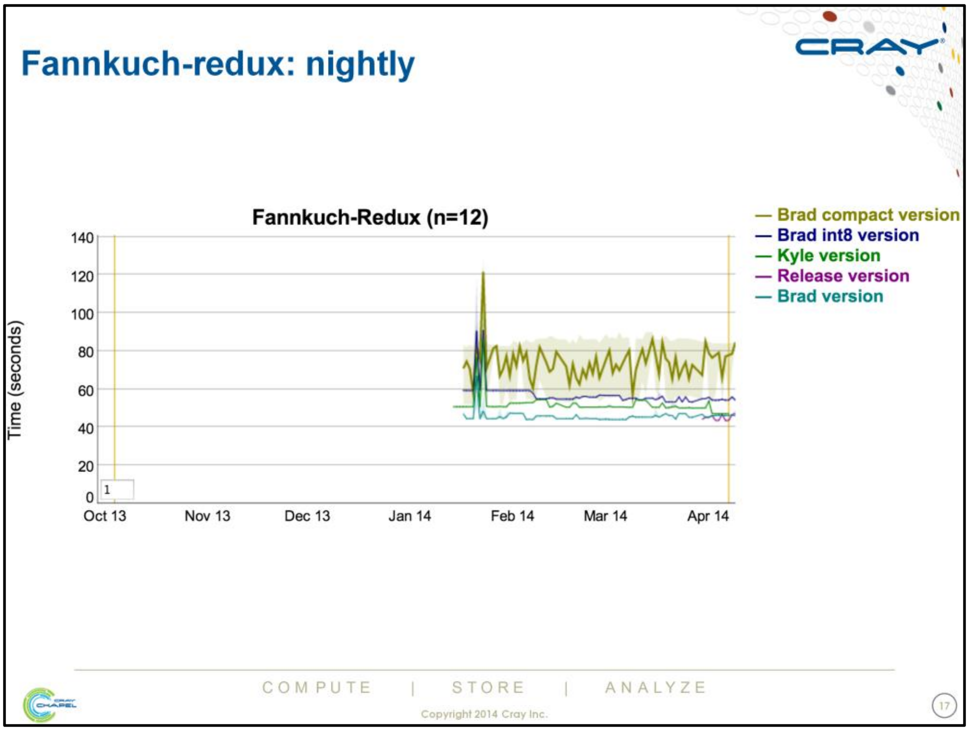
1. Take the first element, X
2. Reverse the first X elements of the sequence
3. Repeat until the first element is 1

# Fannkuch-redux: historical (gathered retroactively)



# Fannkuch-redux: release-over-release






‘Brad compact version’ is noisy due to the use of a reduction

## Fannkuch-redux

### Status:

- The 1.9 release includes a serial version of Fannkuch-redux
  - It performs similarly to most single-threaded reference versions
  - Only a hand-coded SIMD version and multi-threaded versions beat it
  - Not surprisingly, it loses to multi-threaded reference versions

### Next Steps:

- Implement a parallel version of the benchmark
- Simplify our generated code to reduce overheads
  - reference temps seem to be a particular problem
- Investigate how one would write the SIMD version in Chapel





Without a parallel version we will not stack-up against other versions in the multi-core tests.

In analyzing the final optimized assembly I noticed that the C compilers (gcc 4.8 / clang) were turning one of the loops in the reference C version into a memcpy, but not in ours. Getting our loops into a form where the backend compiler will perform this optimization would be a small performance win.

# Spectral-norm

## Spectral-norm

### Overview:

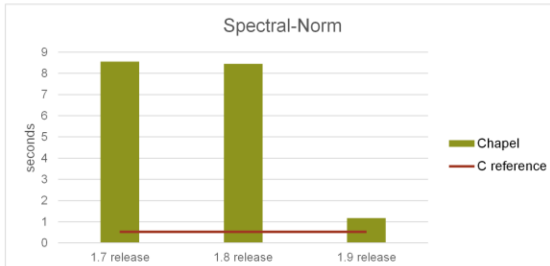
- Calculates the spectral norm of an infinite matrix
  - the square root of the maximum eigenvalue of a square matrix multiplied by its conjugate transpose
  - repeat this computation many times
- Emphasizes the performance of parallel constructs



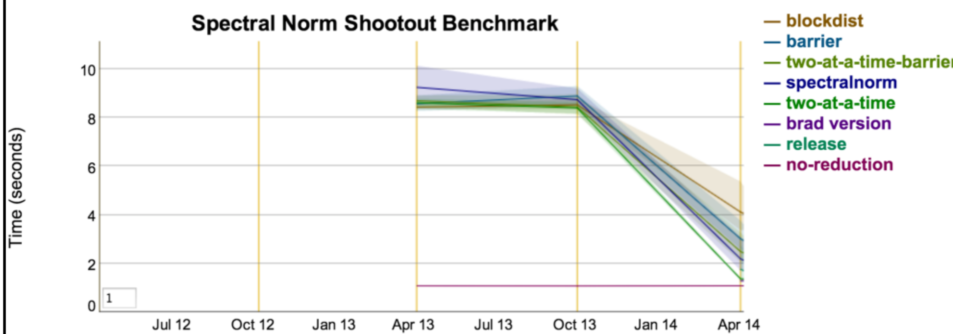

Conjugate transpose is the transposition (A(i,j) => A(j,i)) of a conjugate matrix, where the conjugate of a complex number a + bi is a – bi.

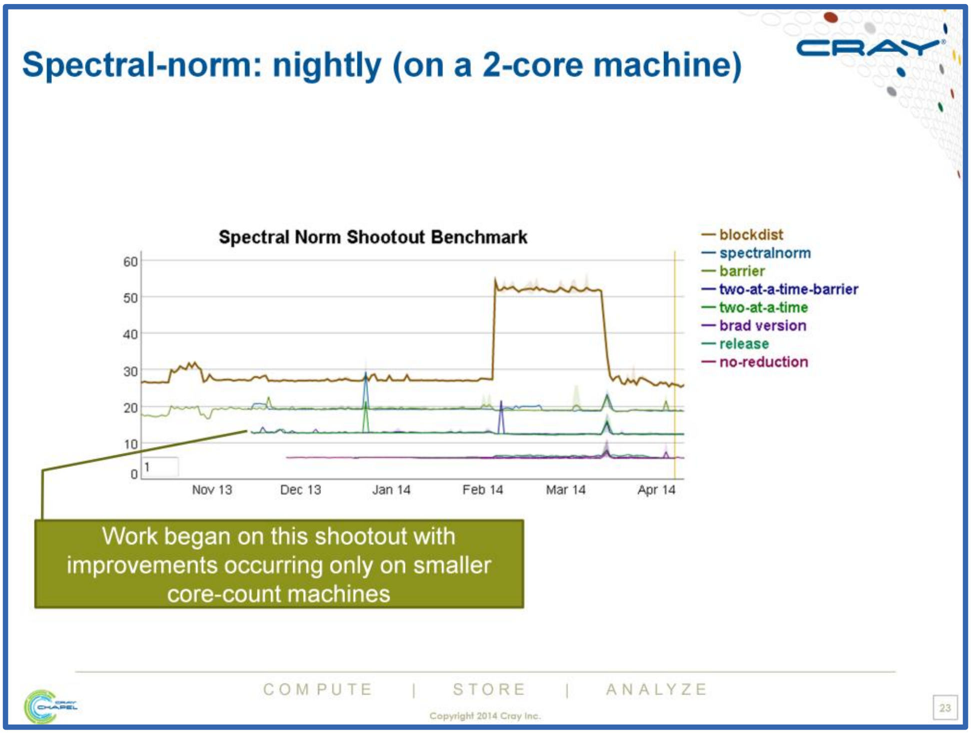
Eigenvalues are a special set of scalars associated with a linear system of equations .

# Spectral-norm: historical



# Spectral-norm: release-over-release

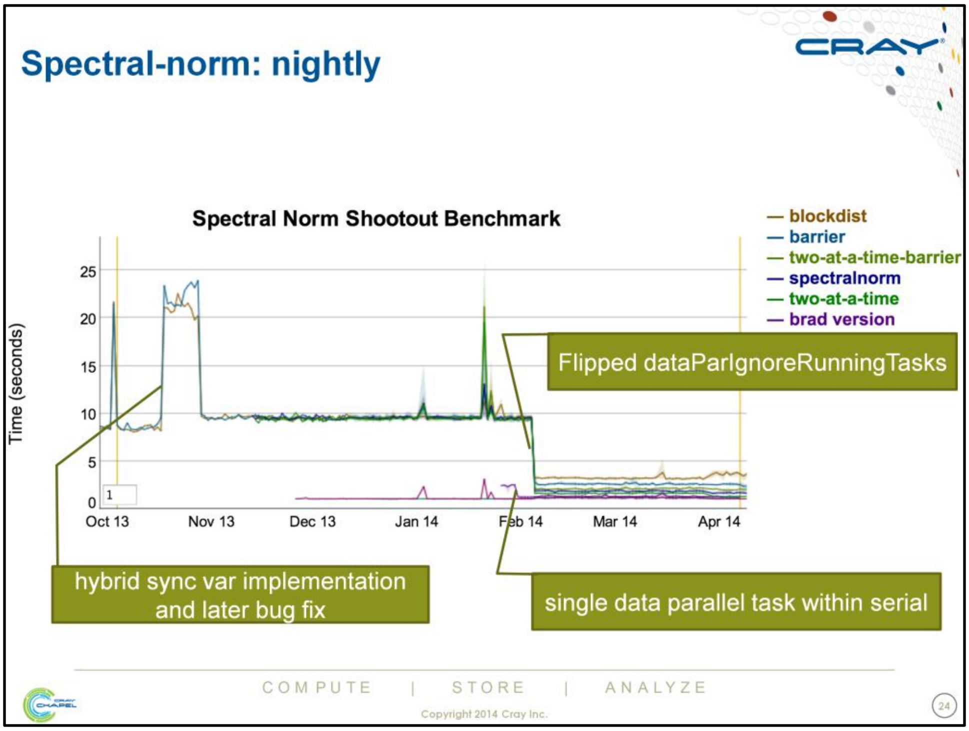





- these results were gathered on chap03, a 2-core workstation
- 'Blockdist' and 'barrier' were the original versions, written by Albert Sidelnik

- The line labeled "spectralnorm" (in light blue, visible on the second set of lines from the top) was an initial cleaned up version, which did not use block distributions or a barrier.

- The two lines labeled “two-at-a-time” and “two-at-a-time-barrier” were based on the gcc #4 reference version, where tasks were created every two iterations instead of every single iteration.




The fastest version shown here involved manually removing nested parallelism. This was motivated by observations related to different task creation policies on small-core-count machines (shown on the next slide). The impact of this result led us to examine – and eventually flip-- the default value of dataParIgnoreRunningTasks. This improved most of the remaining versions which relied on nested parallelism (specifically, a reduction within a forall loop).

The “brad” version squashes the reduction’s parallelism using ‘serial’ statements (and it also included other style changes, including different writes, division instead of bit shifts, formal argument domain query syntax, and alternate methods of array access). This version’s performance improved when we reduced the number of tasks used for data parallel constructs within serial statements.

## Spectral-norm

### Next Steps:

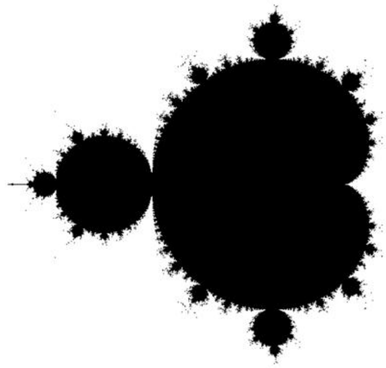
- The benchmark itself is in very good shape
- Remaining performance gap likely due to general Chapel overheads
  - e.g., make Chapel's reduction more competitive with the hand-coded one

# Mandelbrot

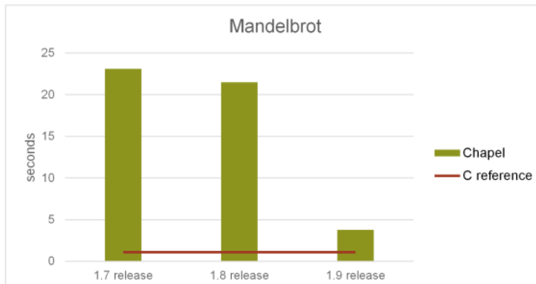
# Mandelbrot

## Overview:

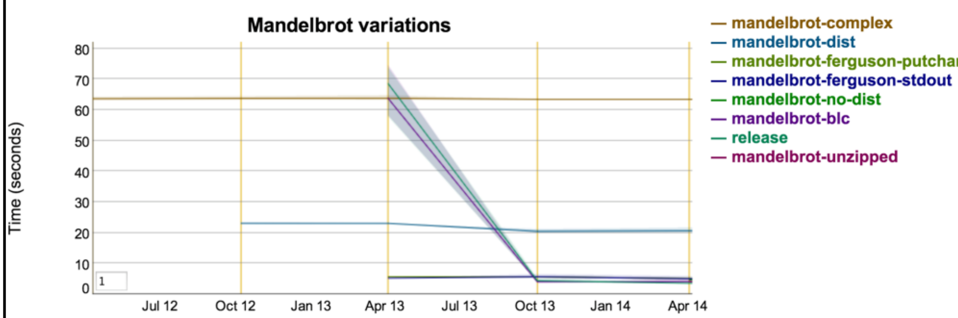
- Computes & plots the Mandelbrot set [-1.5-i, 0.5+i] on an n x n bitmap
- Emphasizes small unsigned integers, multidimensional arrays, and binary output, as well as some bit operations

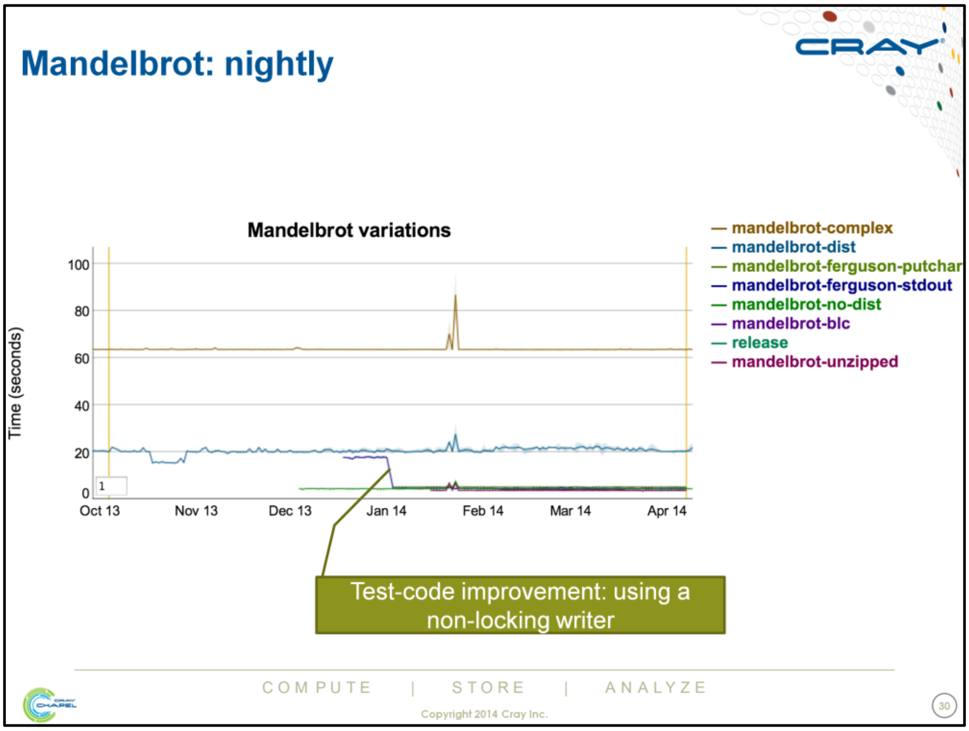


# Mandelbrot: historical

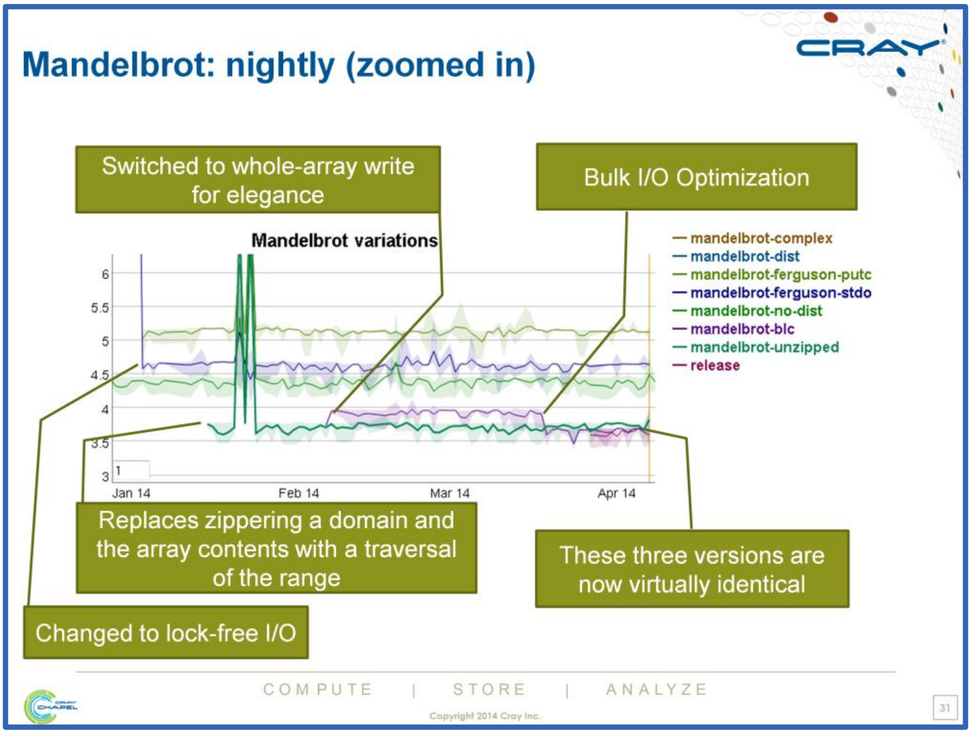


# Mandelbrot: release-over-release






- The 'complex' and 'dist' versions were the original Chapel Mandelbrot versions written by Jacob Nelson. They were very slow, 19x – 40x slower than the reference version and are shown here to emphasize the improvement made by all subsequent versions. 'dist' used a Block distribution, setting it up for distributed memory execution, but adding overhead for the shared-memory shootout competition. The 'complex' version uses complex types and math rather than scalar floating point values.
- The 'no-dist' version is based on a version named 'mandelbrot-fancy', also developed by Jacob, but which ran out of memory




The original versions of the benchmark did not make use of Chapel I/O

The ‘blc’ version is essentially a cleaned-up version of mandelbrot-unzipped.

Not shown is the no-local improvement generated by using the bulk array write

# Mandelbrot

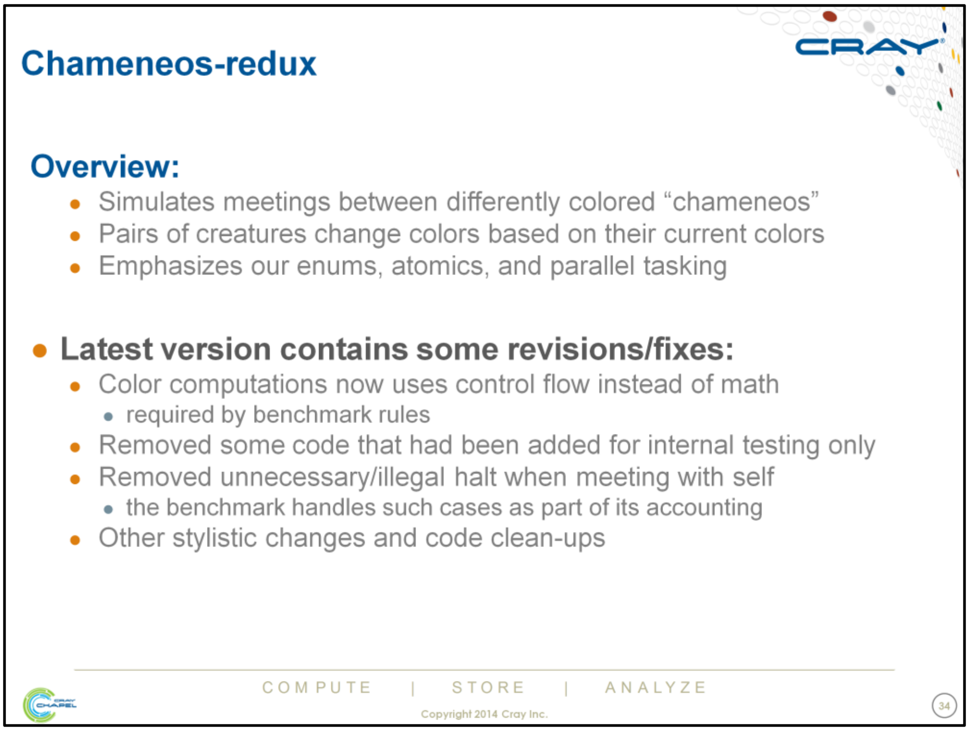
## Next Steps:

- Determine the cause of remaining ~3.5x performance gap
- Consider strength reduction for divisions by powers of two
  - or ensure that the C compiler will do this for us

## Future Work:

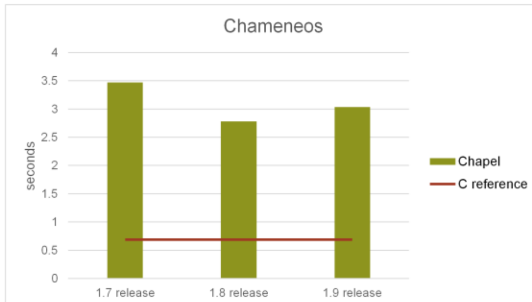
- Explore use of complex types/operations rather than uints
  - What optimizations would be required to minimize differences?

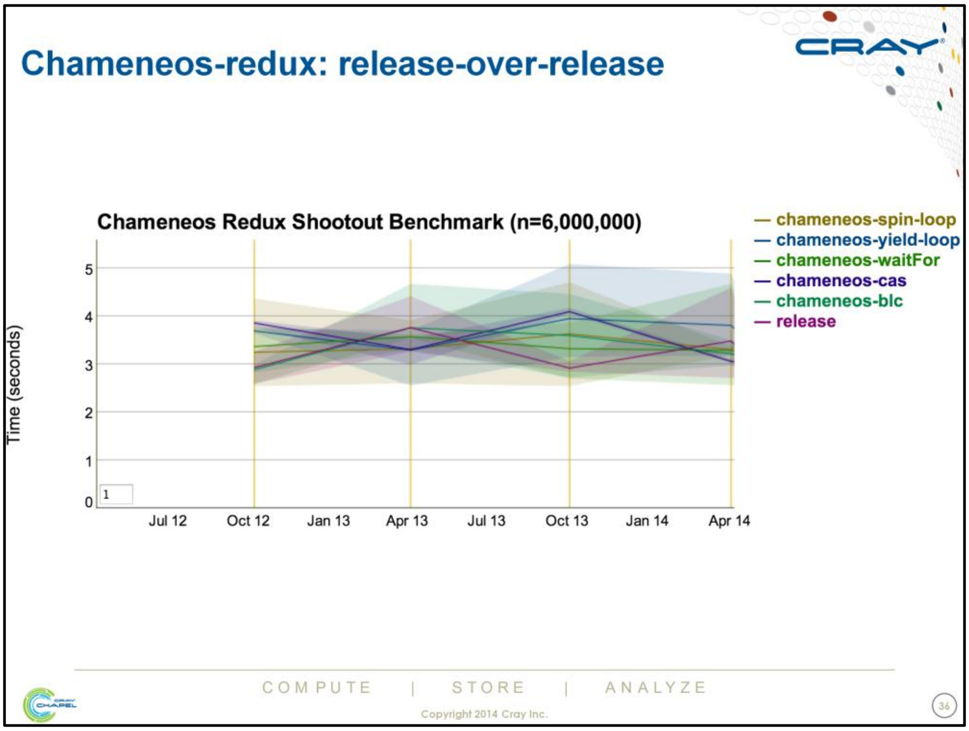
# Chameneos-redux




The stylistic improvements alluded to include removing redundancy while simplifying the code, moving the ownership of certain procedures, and converting some variables to constants.

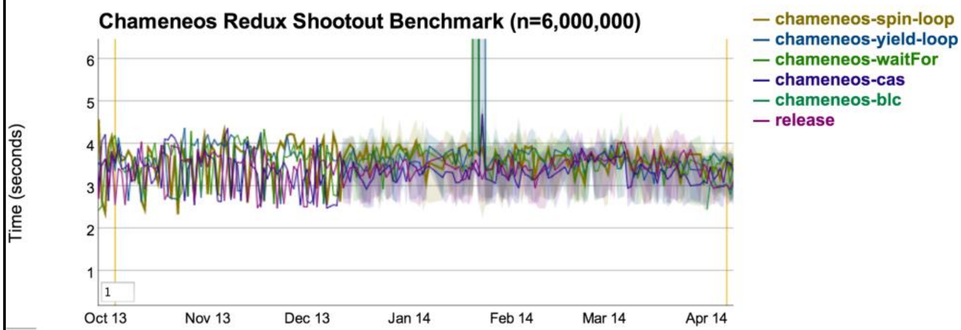
# Chameneos-redux: historical

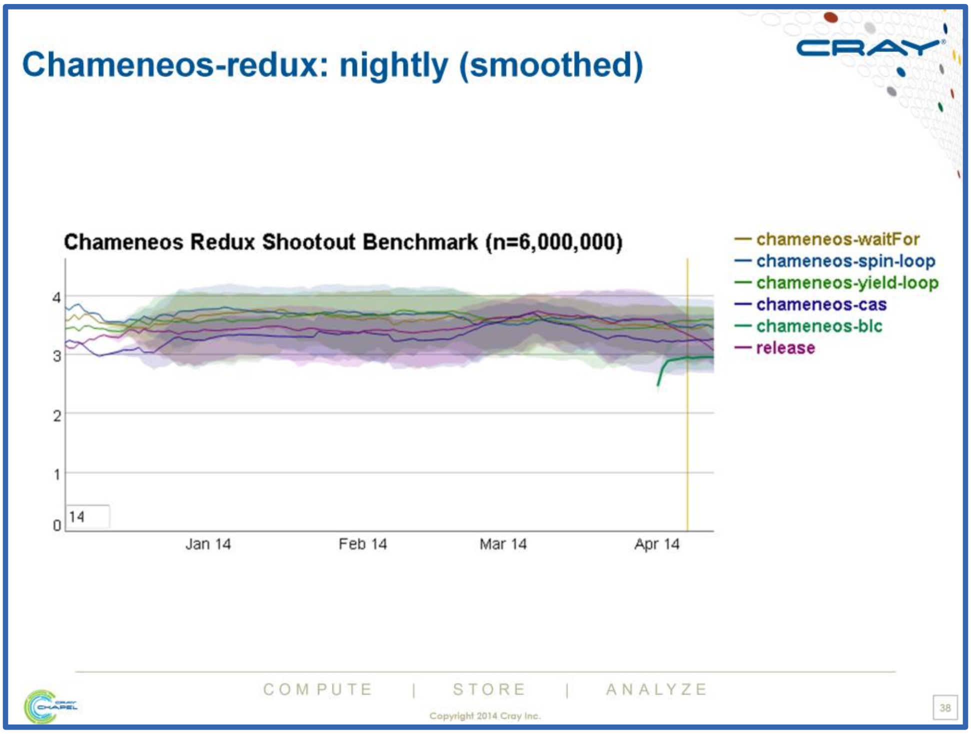





Note: nightlies show nearly no change between 1.8-9

# Chameneos-redux: nightly





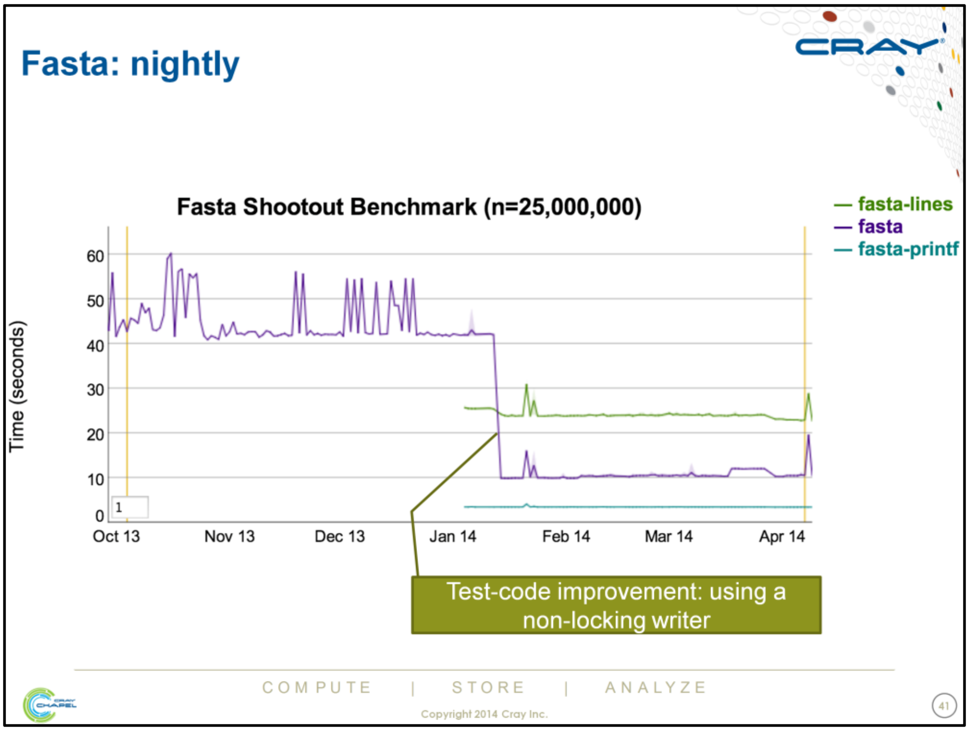

Note: data has been smoothed to increase clarity of actual trends However, due to the high variability of data in these versions, some trends are visible without necessarily meaning anything. The envelopes provided overlap almost completely, indicating that while a test may on average be faster than others, it will not necessarily be the fastest each night. The only notable exception is the new version of chameneos. The gentle slope seen for the red line hides the sharp drop experienced when the release version was converted to adopt code from chameneos-blc.

## Chameneos-redux

### Next Steps:

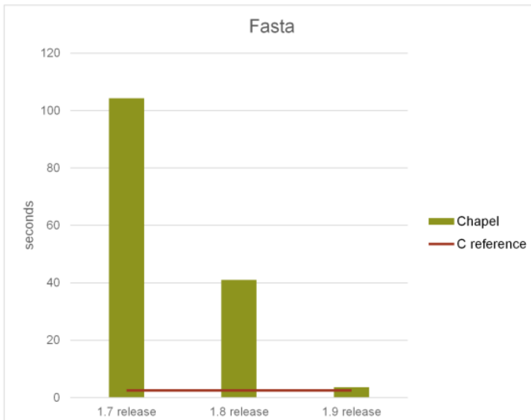
- Make Qthreads tasking the default and check impact on performance
- Find and optimize overheads in the scalar code paths

# Fasta

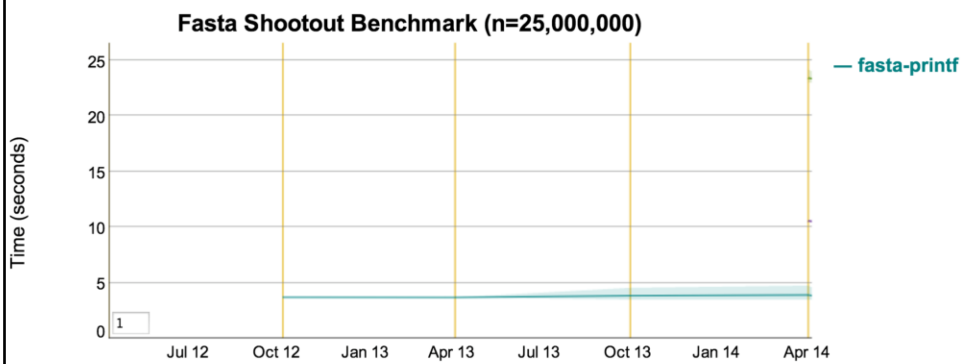



This is the only real change to the fasta code since the last release – both in terms of benchmark code changes and performance code changes.

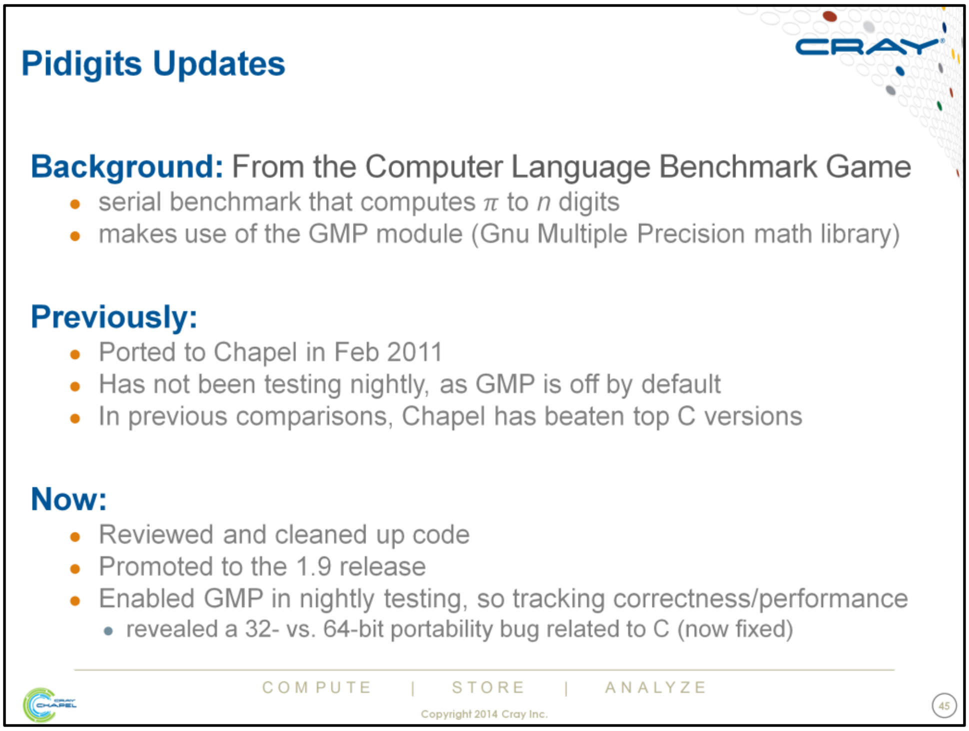
# Fasta: historical



## Fasta: release-over-release



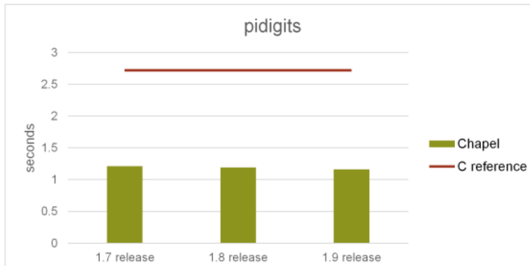
# Pidigits

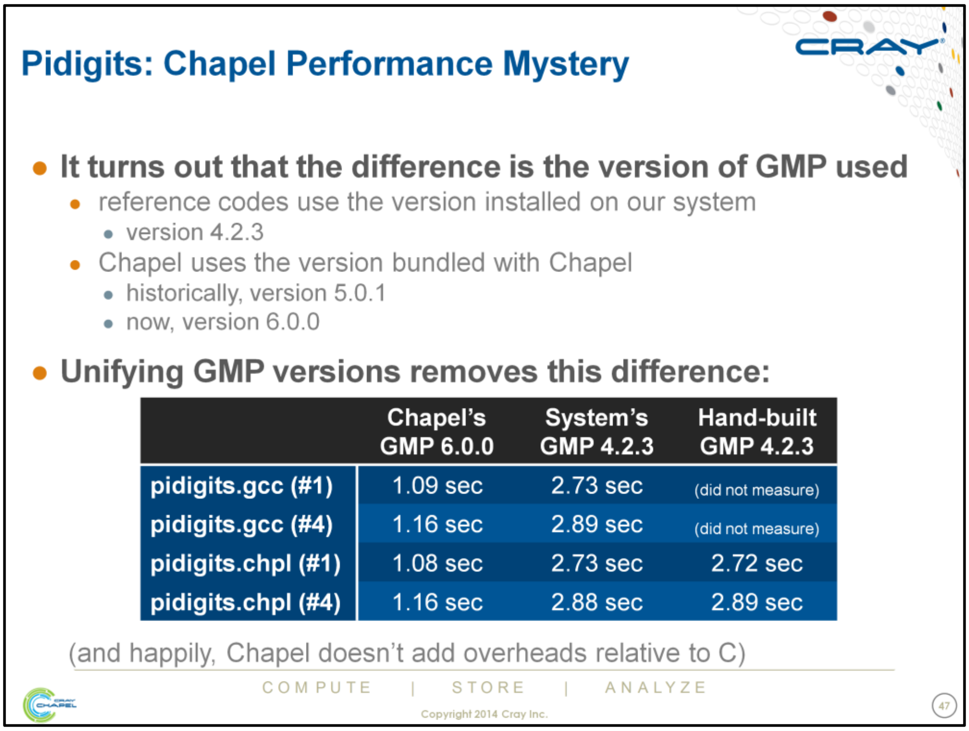



The fact that Chapel has traditionally beat the top C versions in our comparisons seemed suspicious, but we hadn't taken the chance to investigate until now (see following slides)

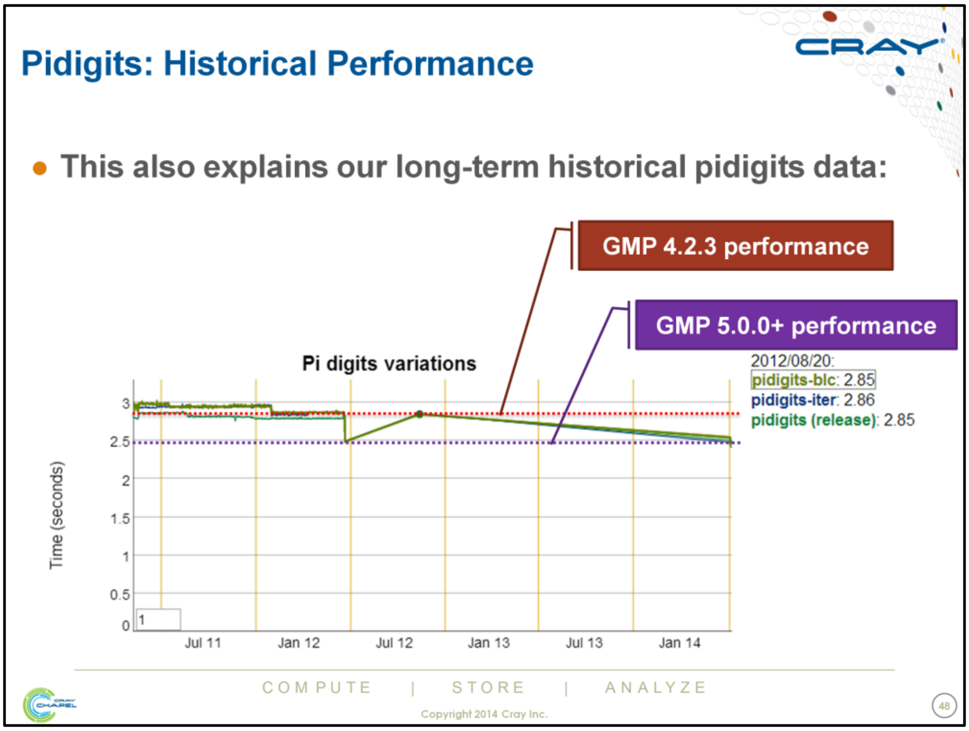
# Pidigits: historical

- Traditionally, pidigits in Chapel has been surprisingly fast:

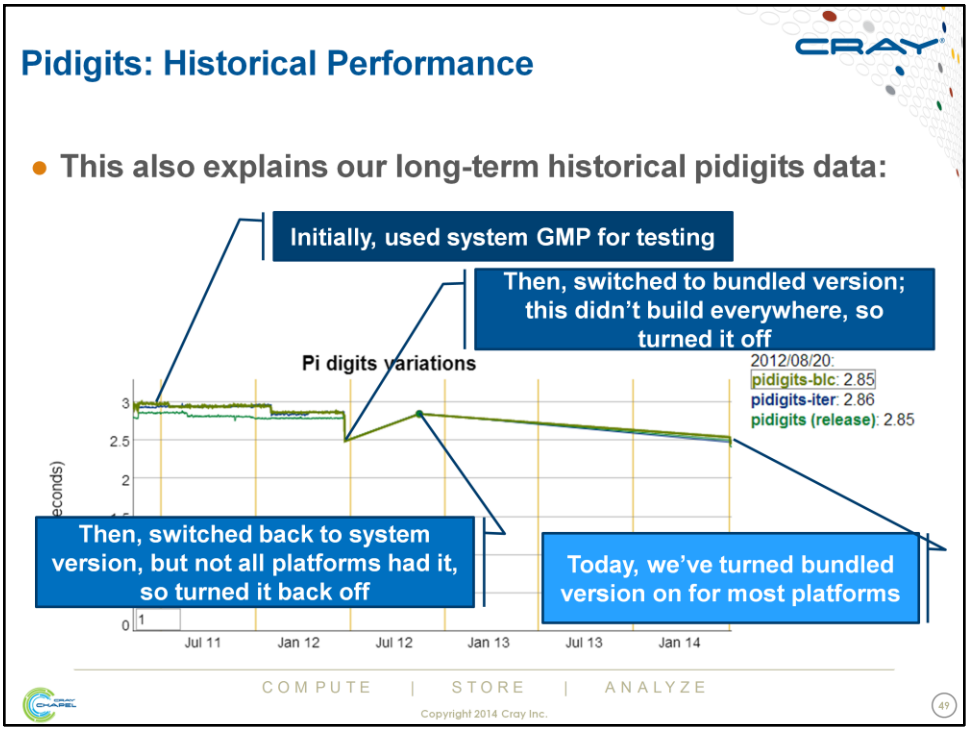





The C versions weren't measured against the hand-built GMP versions mostly out of laziness – it's slightly painful to override the system version of GMP and the trends were pretty clear from these measurements.




These timings are from a different machine than the previous slide; that's why the numerical values don't match.




These timings are from a different machine than the previous slide; that's why the numerical values don't match.

## Pidigits Next Steps: defaults in releases

- **Decide whether to build GMP by default**
  - Possible proposal:
    - try to build, ignoring failures
    - current CHPL_GMP logic will default to bundled version if build worked
    - main downside: won't default to using system version (same as today)

- **Decide whether to switch to bundled version for Crays**
  - today we're using the system version on Crays by default
    - this turns out to be 4.2.3
    - so, bundled version could be better

## Pidigits Next Steps: reduce reliance on C types

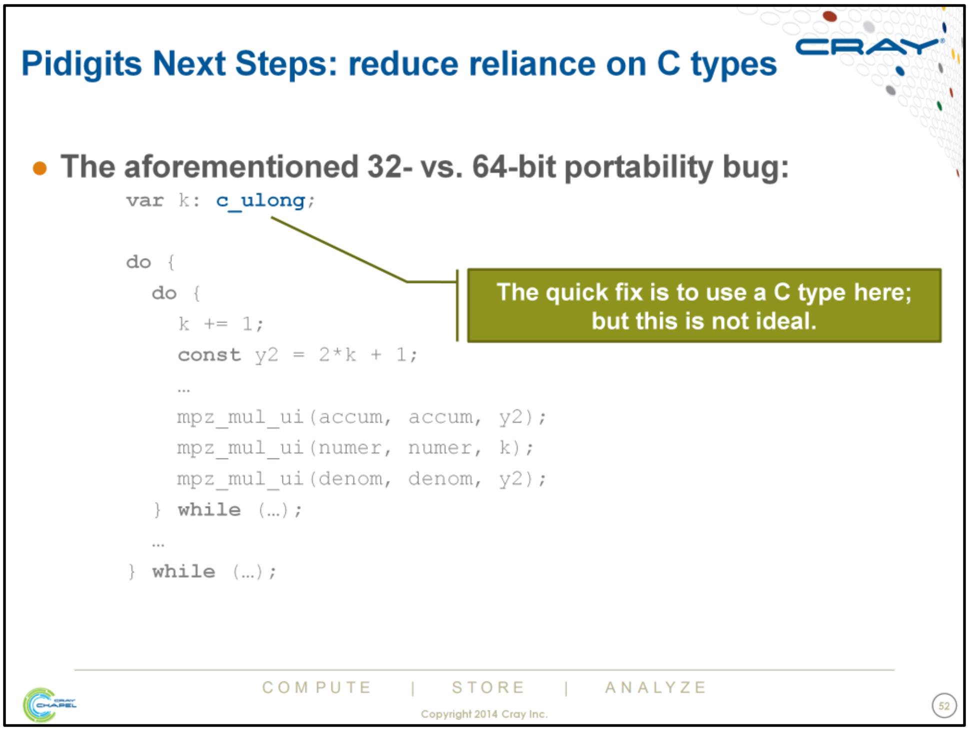
- **The aforementioned 32- vs. 64-bit portability bug:**

```
var k: uint;

do {
  do {
    k += 1;
    const y2 = 2*k + 1;
    …
    mpz_mul_ui(accum, accum, y2);
    mpz_mul_ui(numer, numer, k);
    mpz_mul_ui(denom, denom, y2);
  } while (…);
  …
} while (…);
```

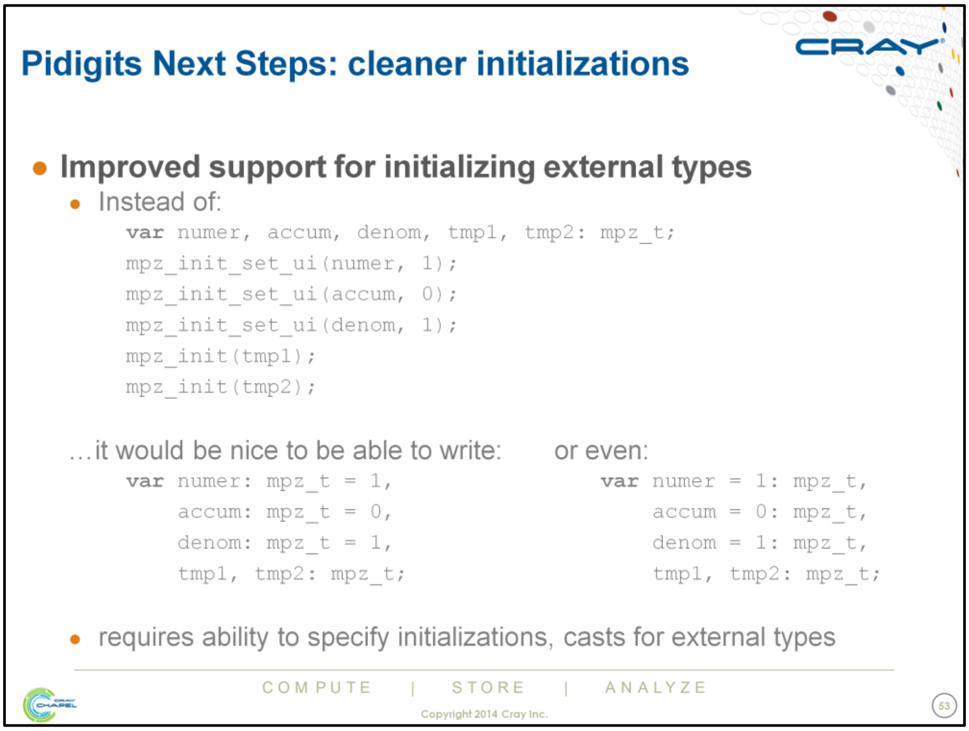
Problem: mpz*_ui routines expect a C unsigned long.

Chapel's uint type is 64 bits and not guaranteed to fit in a C unsigned long




Not ideal because if, to use GMP, you have to use C types everywhere, what does that imply for your Chapel code?

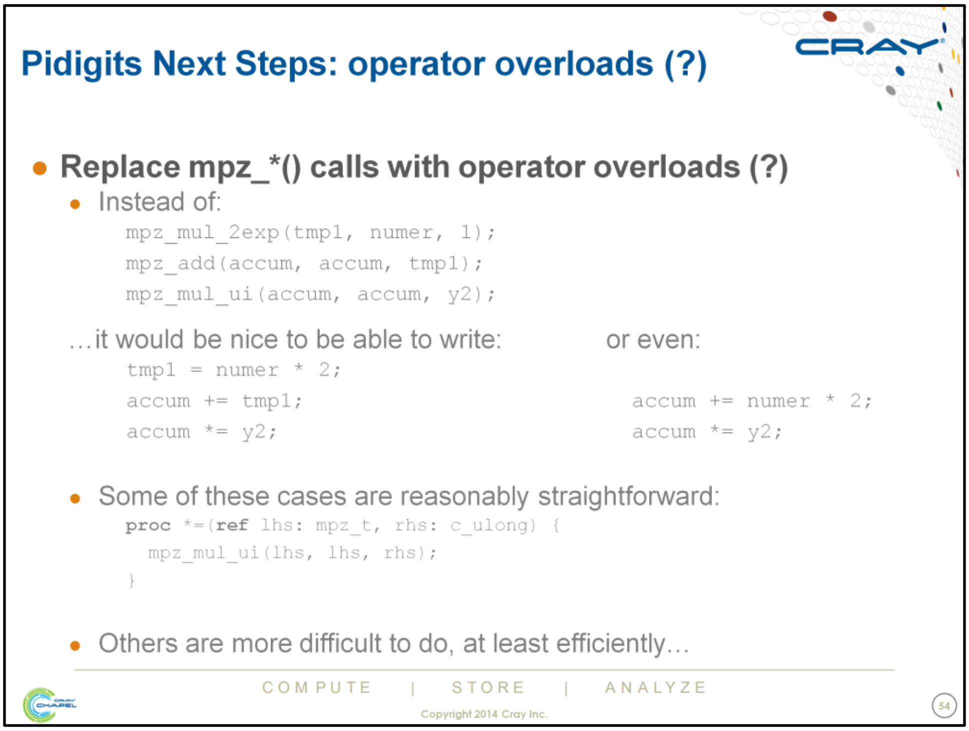
The primary alternative would be to have Chapel's GMP routines downcast its int arguments to the appropriate C types; but at what cost/risk? Could, for example, have a safer but more expensive vs. cheaper and more risky mode which is guided by a --fast-controlled flag.




More specifically, what we might want/need to support this are:

- promotion of initialization assignments to a language-level concept
- and/or the ability to define a defaultInitialize() function on a specific type when its initializer isn't present
- and the promotion of user-defined casts to a language-level concept

In the cast case, there's also a challenge related to the desire to take the mpz_t that I'm imagining would be created and returned by the cast function and steal it for use by 'numer' rather than requiring a copy from one mpz_t to another




The challenges alluded to in the final bullet here relate to the fact that to use GMP best, you'd really want to recognize and match against multi-expression templates.

Failure to do so requires extra temporary variables that would either have to be reference counted or leaked.

But how to support such multi-expression templates for external types that the compiler doesn't know about or know how to reason about?


Why are mpz_t overloads ambiguous with existing operators?  Because mpz_t types are 1-element arrays in C and for that reason are represented as 1-tuples in Chapel, conflicting with our tuple overloads.

The compiler-introduced temps bullet refers not only to the fact that the compiler's inserting such temps (which we probably don't want), but also to the fact that such temps are not l-values, yet most GMP functions currently take their arguments by ref. This could potentially be resolved by changing such read-only GMP arguments to take their arguments by const ref – I haven't tried that yet.

# Pidigits Next Steps: op overloading promise

- **Despite the challenges, the potential is tantalizing:**

```
var numer, accum, denom, tmp1, tmp2: mpz_t;
mpz_init_set_ui(numer, 1);
mpz_init_set_ui(accum, 0);
mpz_init_set_ui(denom, 1);
mpz_init(tmp1);
mpz_init(tmp2);
do {
  do {
    mpz_mul_2exp(tmp1, numer, 1);
    mpz_add(accum, accum, tmp1);
    mpz_mul_ui(accum, accum, y2);
    mpz_mul_ui(numer, numer, k);
    mpz_mul_ui(denom, denom, y2);
  } while (mpz_cmp(numer, accum) > 0);
  mpz_mul_2exp(tmp1, numer, 1);
  mpz_add(tmp1, tmp1, numer);
  mpz_add(tmp1, tmp1, accum);
  mpz_fdiv_qr(tmp1, tmp2, tmp1, denom);
  mpz_add(tmp2, tmp2, numer);
} while (mpz_cmp(tmp2, denom) >= 0);
const d = mpz_get_ui(tmp1);
mpz_submul_ui(accum, denom, d);
mpz_mul_ui(accum, accum, 10);
mpz_mul_ui(numer, numer, 10);
```

**vs.**

```
var numer = 1: mpz_t,
    accum = 0: mpz_t,
    denom = 1: mpz_t,
    tmp1, tmp2: mpz_t;


do {
  do {

    accum += (numer * 2);
    accum *= y2;
    numer *= k;
    denom *= y2;
  } while (numer > accum);
  tmp1 = numer * 2;
  tmp1 += numer;
  tmp1 += accum;
  (tmp1, tmp2) = divmod(tmp1, denom);
  tmp2 += numer;
} while (tmp2 >= denom);
const d = tmp1: c_ulong;
accum = (accum - denom)*d;
accum *= 10;
numer *= 10;
```

## Pidigits Next Steps: multi-precision types?

- At which point, maybe we'd want to promote arbitrary-width types to the language?

instead of…

```
var numer = 1: mpz_t,
    accum = 0: mpz_t,
    denom = 1: mpz_t,
    tmp1, tmp2: mpz_t;
```

use…

```
var numer = 1: real(*),
    accum = 0: real(*),
    denom = 1: real(*),
    tmp1, tmp2: real(*);
```







Note that if we went directly to this approach, it would allow us to dodge several of the previous challenges.

Yet, the downside to doing so is that other user-defined external types would not enjoy these benefits.

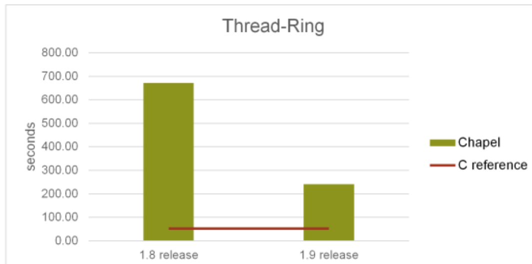
# Thread-ring

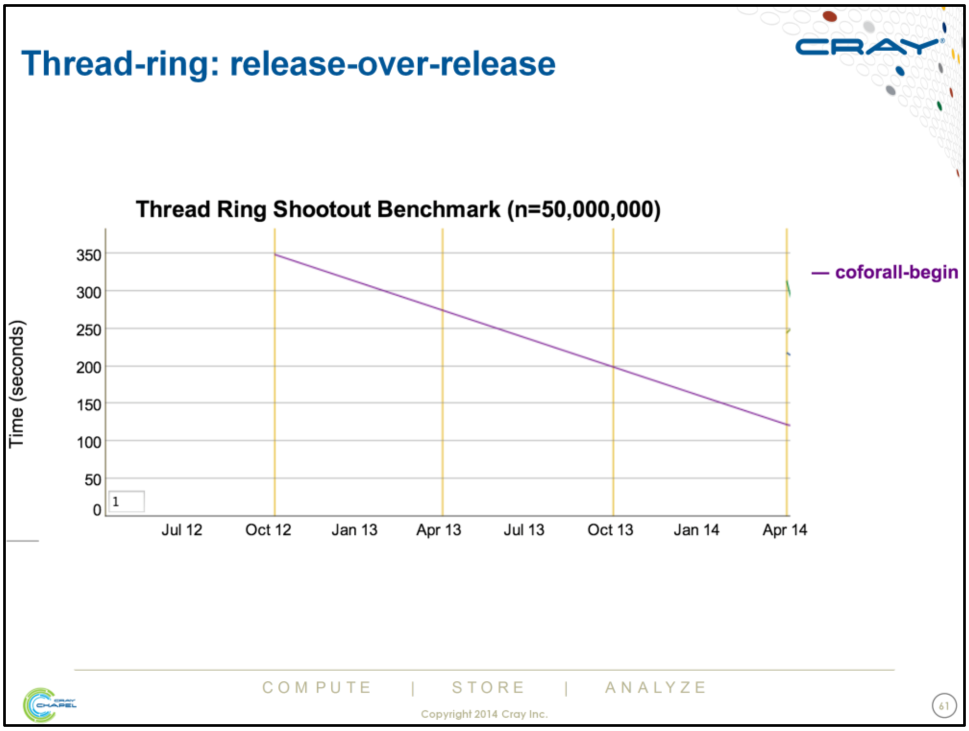
# Thread-ring

## Overview:

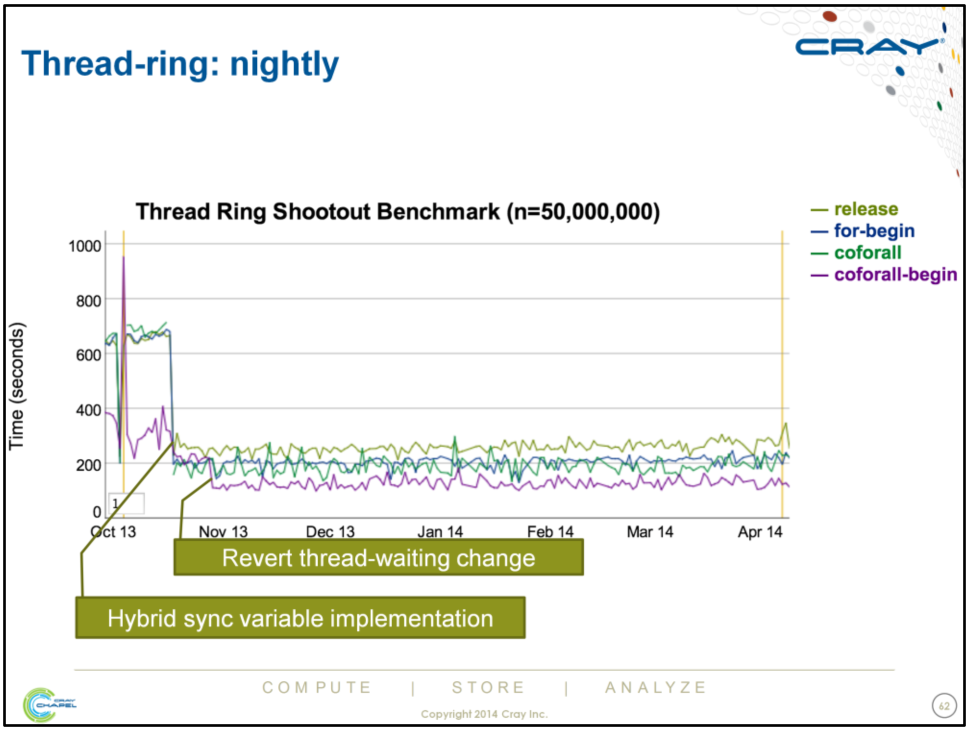
- Passes a token *n* times among *nthreads* threads
  - prints the thread that ends up holding the token
- Problem Size: n = 50,000,000, nthreads = 503
- Emphasizes tasking and synchronization variables

# Thread-ring: historical






Most of the historical timings time out for the four different versions. One of the three runs of the Oct12 execution happened to not time out which is why there is one line with two data points. This graph primarily shows that the versions have gone from (typically) timing out to completing in version 1.9.




Using begin statements within a coforall loop is still the fastest way to implement this program, although the other methods are not as far behind as they used to be. It was theorized that leftover threads were being repurposed instead of initializing new ones, leading to the timing difference. Testing prior to the sync variable change seemed to confirm that theory, although no tests were performed after that change and the change back to the old thread-waiting version.

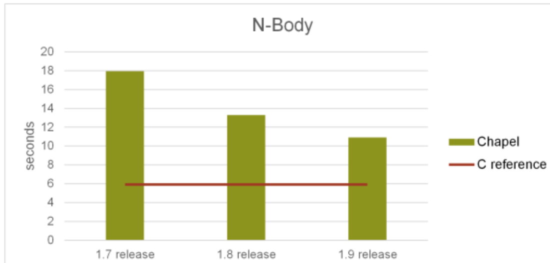
## Thread-ring

### Next Steps:

- Work on making Qthreads the default to get lower tasking overheads
- Map sync vars to Qthreads' full-empty variables
- Identify and fix remaining performance gaps compared to references
- Consider additional optimizations to 'fifo' tasking and sync vars

# N-body

## N-Body: Historical



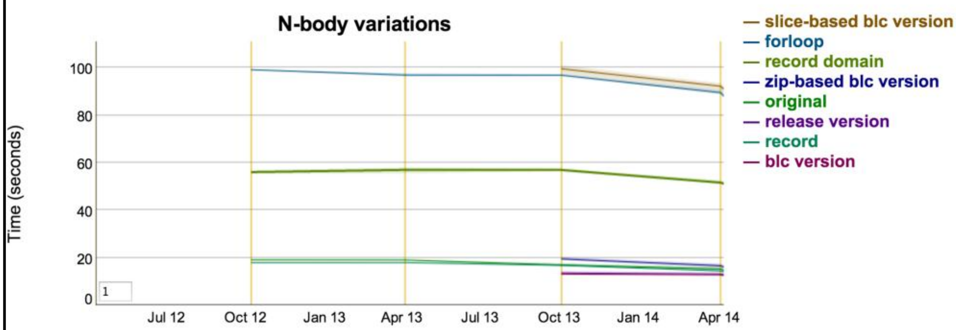

COMPUTE | STORE | ANALYZE



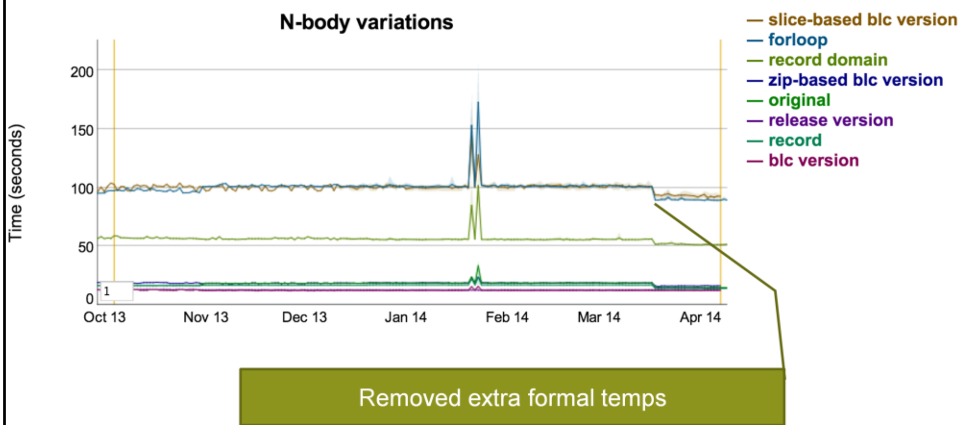


From this point out, less and less has been done with the benchmarks themselves, so we report less on work done and simply present the performance graphs and key changes.

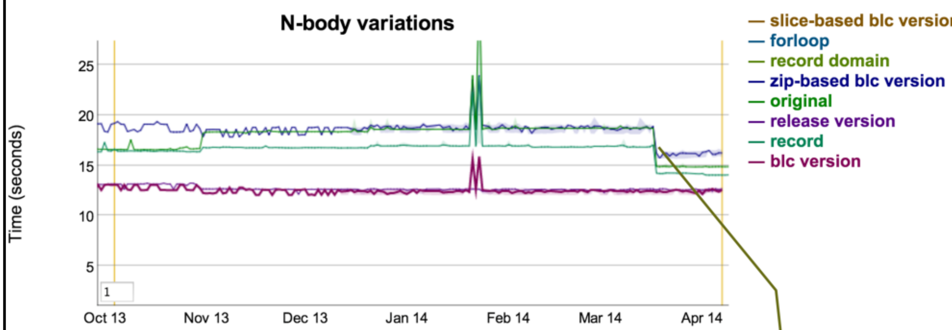
## N-Body: all versions



# N-Body: Nightly results



# N-Body: best versions

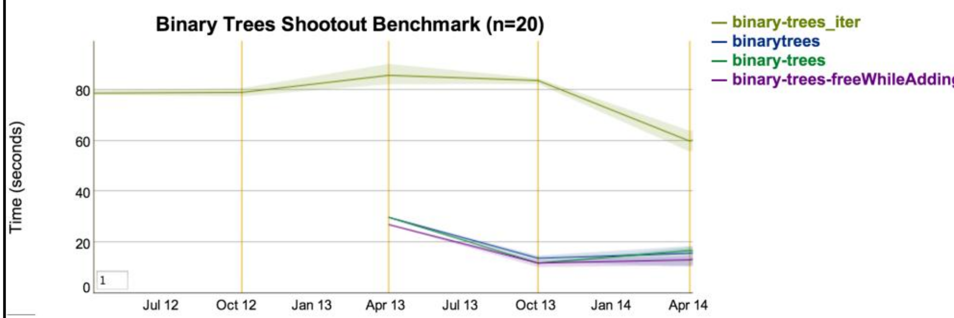



Removed extra formal temps

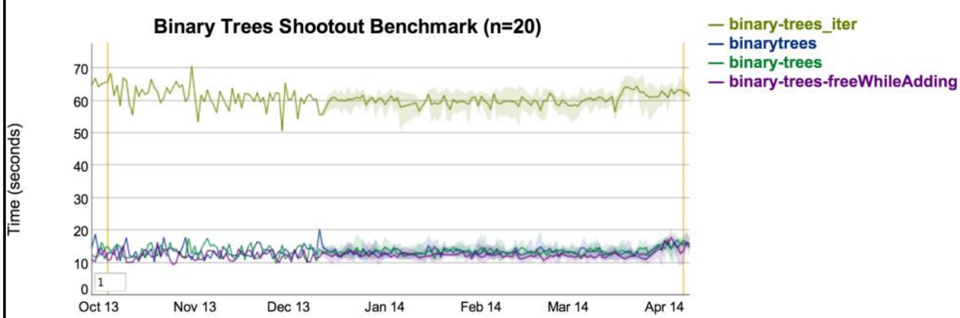
# Binary-trees

# Binary-trees: historical

# Binary-trees: release-over-release

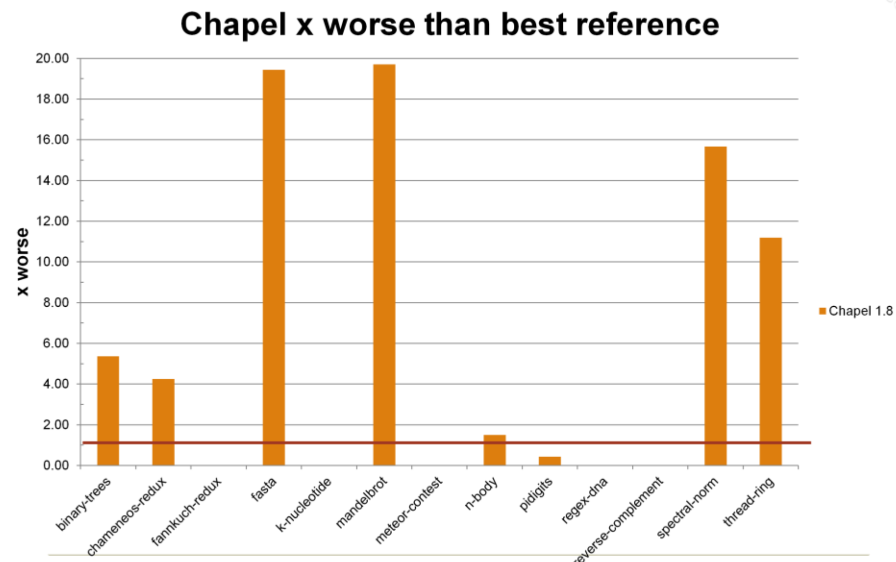


# Binary-trees: nightly

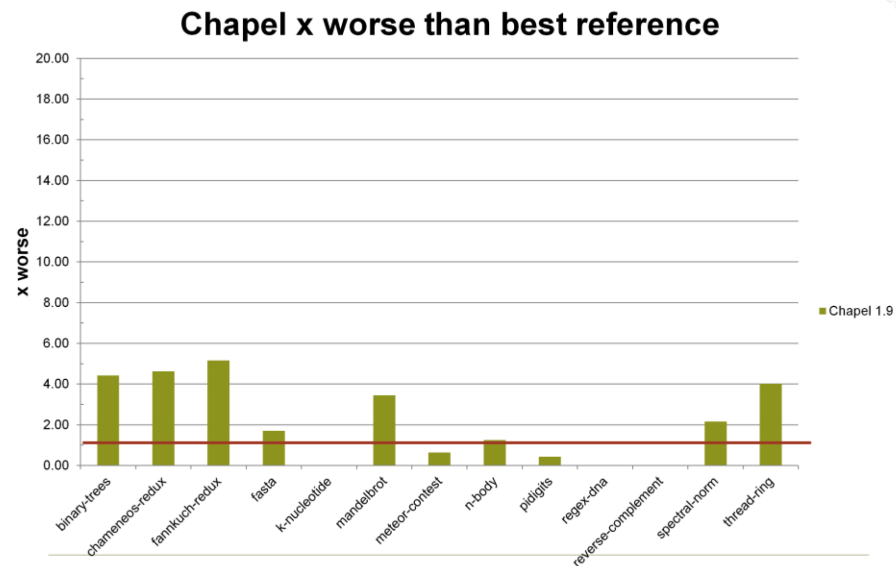


# Shootout Summary

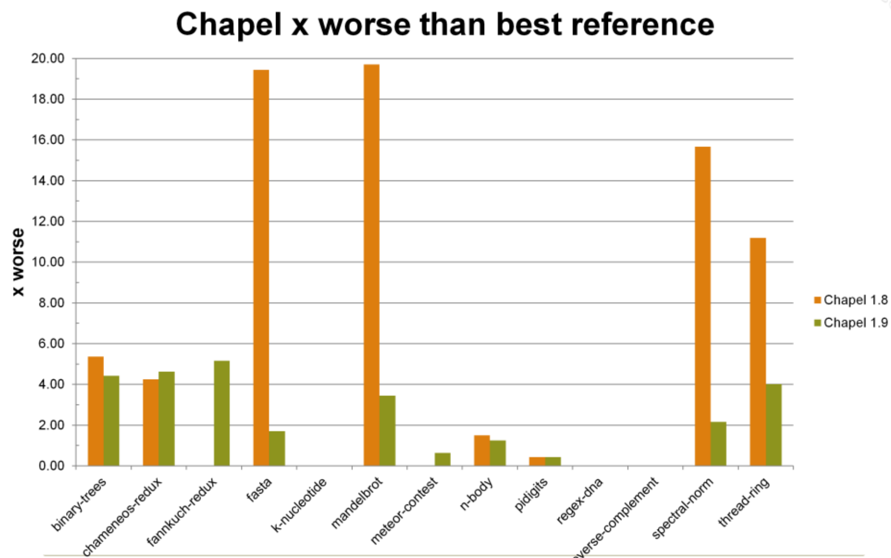
## Shootout Performance Summary (v1.8)



# Shootout Performance Summary (v1.9)



## Shootout Performance Summary (v1.8-v1.9)




COMPUTE | STORE | ANALYZE

# Outline

- Computer Language Benchmark Game ("shootout") Codes
- Other Notable Single-Locale Benchmark Results
  - LULESH
  - MiniMD
  - SSCA#2
  - HPC Challenge Benchmarks
  - NAS Parallel Benchmarks

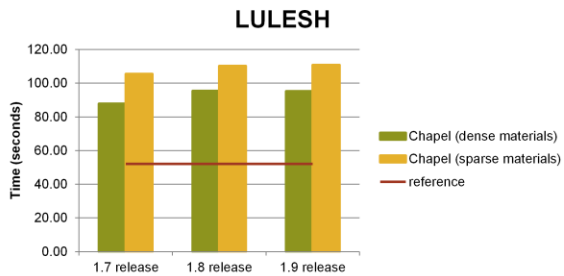
## Other Notable Benchmark Results

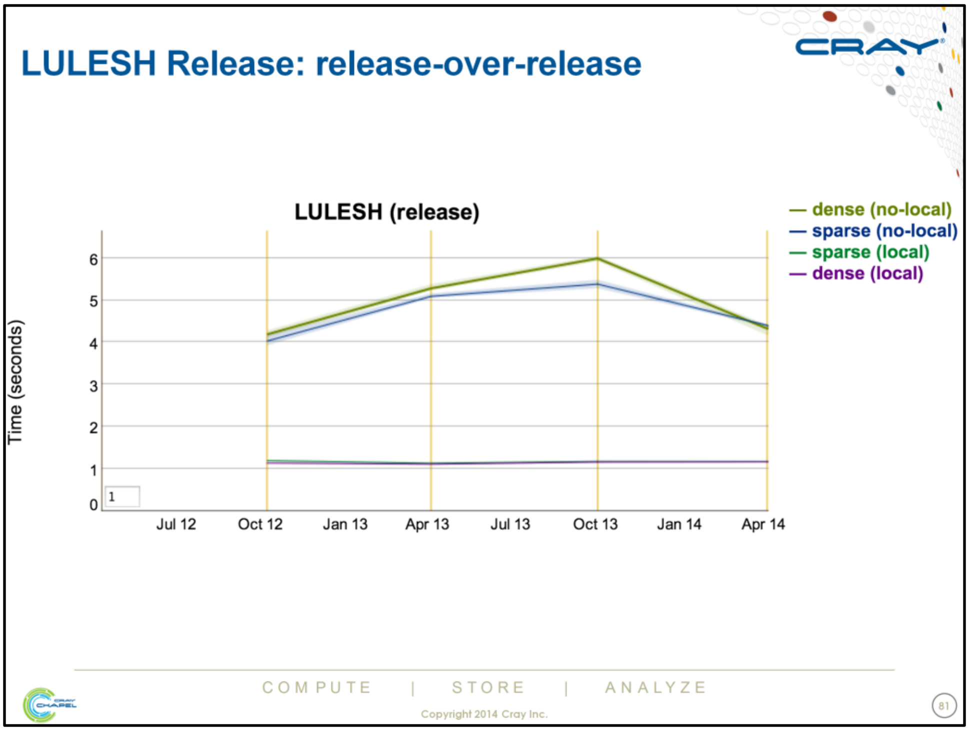
- **The following benchmarks are important to us, yet have not received much attention in the 1.9 release cycle**
  - we've focused on the shootouts to fix serial performance issues
  - these will be more compelling for HPC users than the shootouts
  - graphs show that they've generally improved in spite of this
    - though there are a few performance slips as well

- **These are all multi-locale benchmarks**
  - the test results here represent single-locale executions
    - some cases were compiled --no-local to track multi-locale code overheads

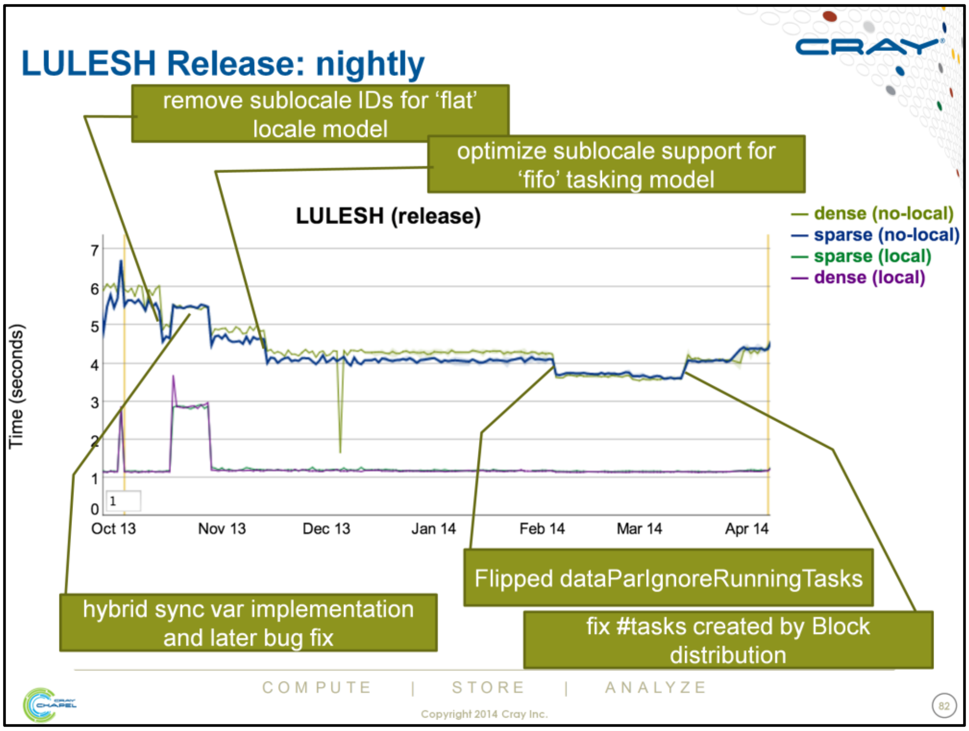
# LULESH

# LULESH: historical





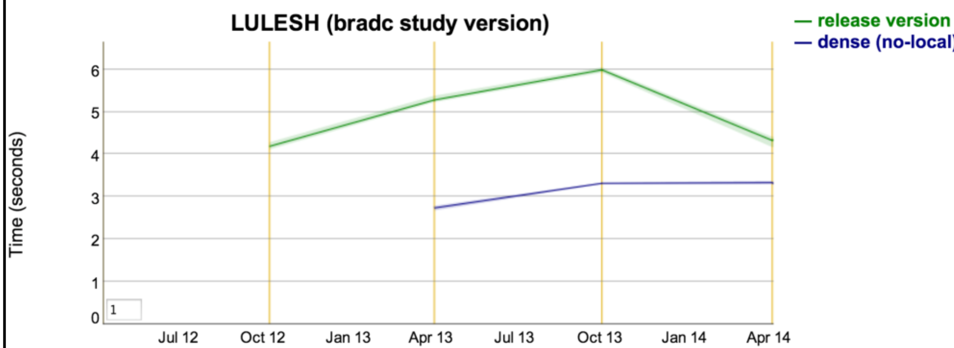

Note that the previous slide showed –local timings only. The –no-local cases got better during this release cycle.



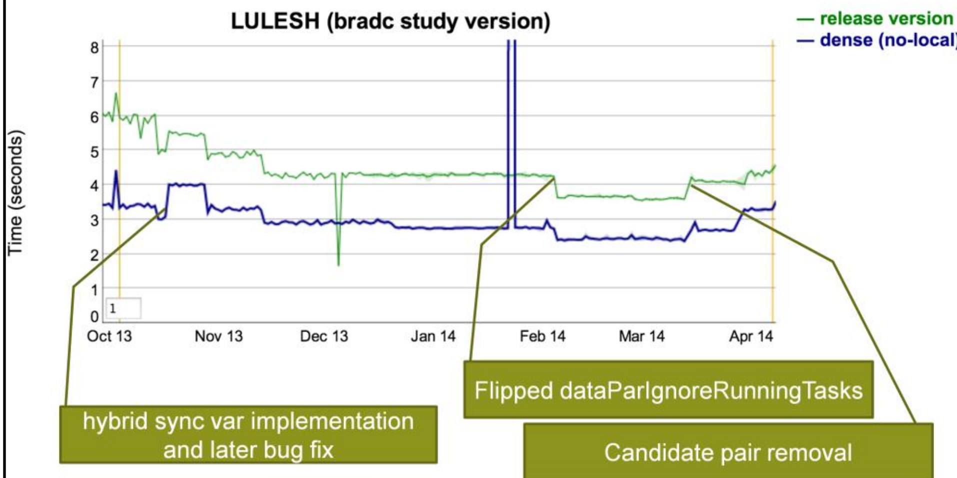

We haven't had the chance to check what the final performance regression in LULESH is due to. The most likely candidates are:

- we started using the –static flag for these performance tests
- we moved the task counting from the runtime to the module
- we changed some 'inout' intents to 'ref' intents in the module code (but primarily for I/O which seems unlikely to be the cause here)

# LULESH Study: release-over-release

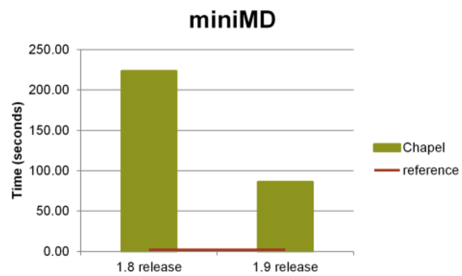


# LULESH Study: nightly

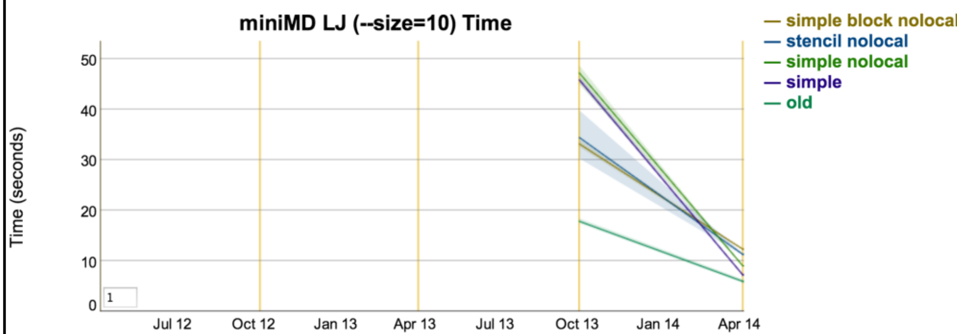


# miniMD

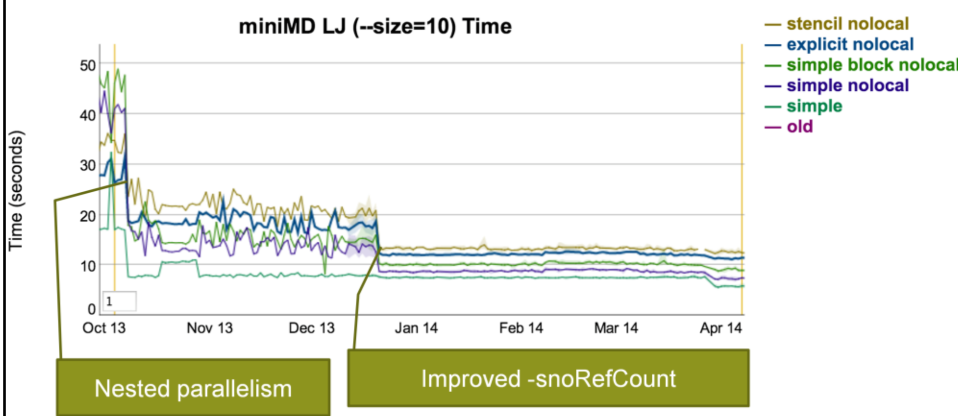
# miniMD: historical



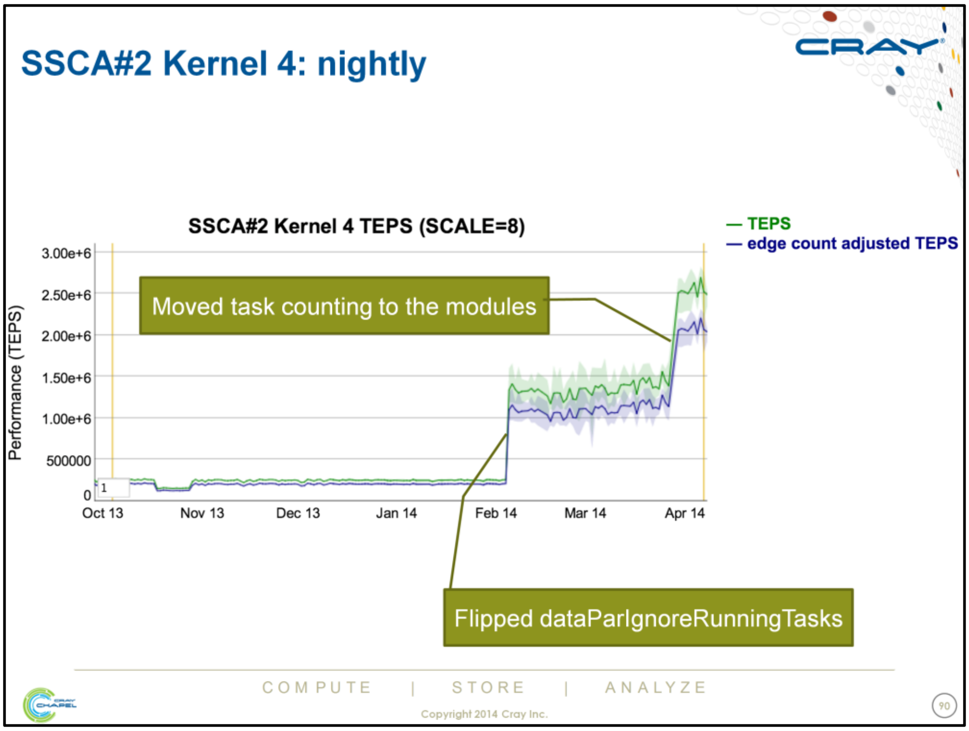
# miniMD: release-over-release



# miniMD: nightly

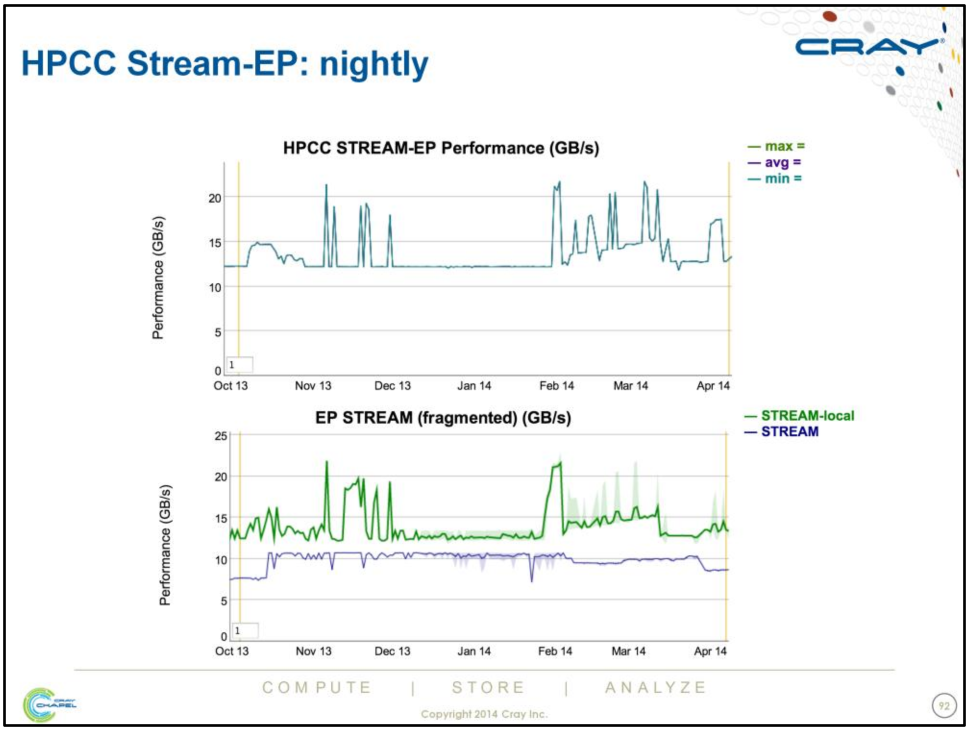


# SSCA#2

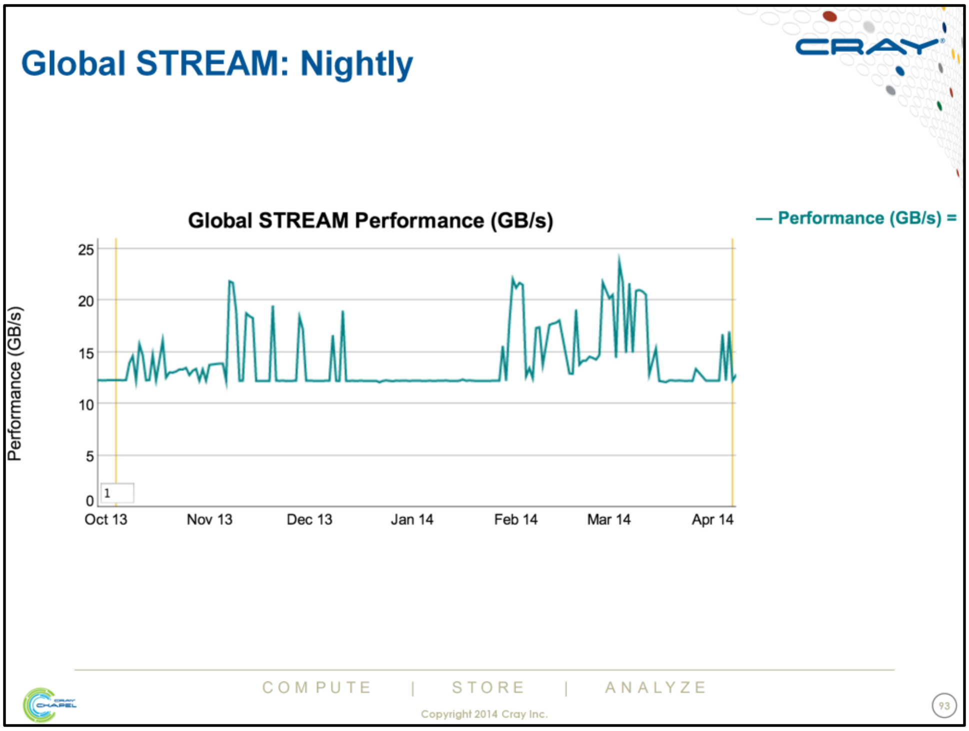



Note that this is a performance slide, and that therefore higher is better.

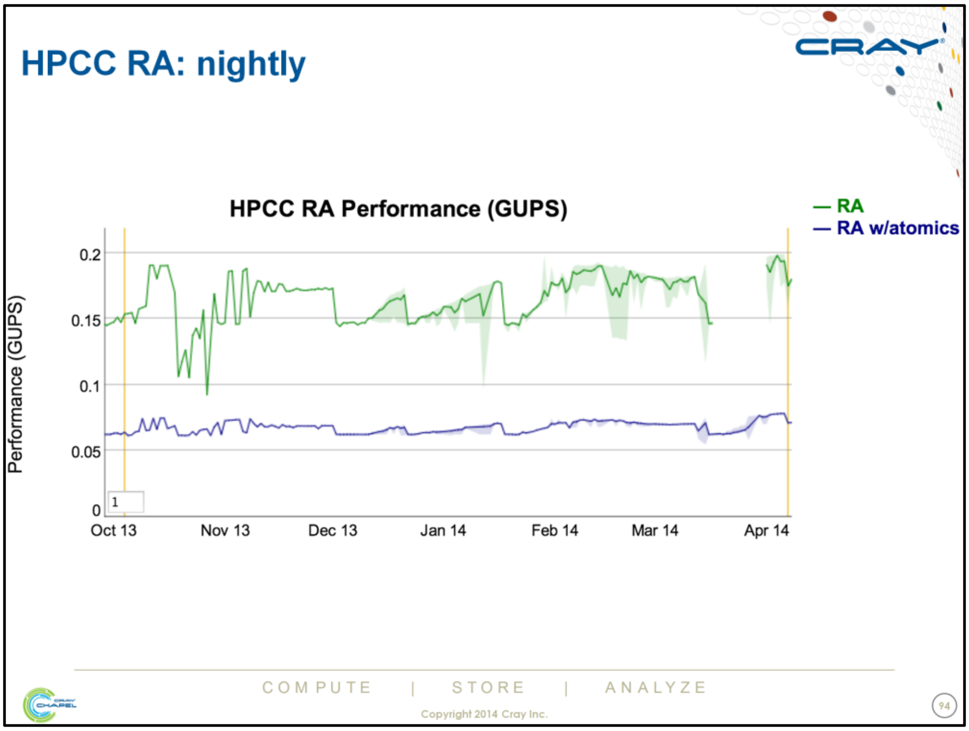
## HPC Challenge Benchmarks



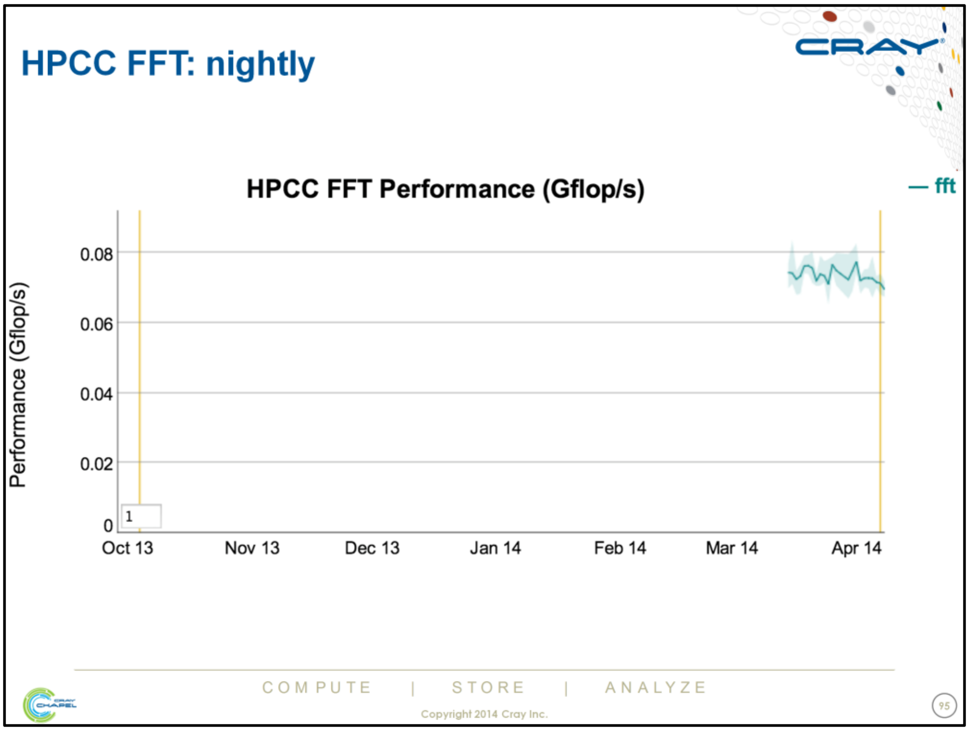

Note that these are performance slides and that higher is better.



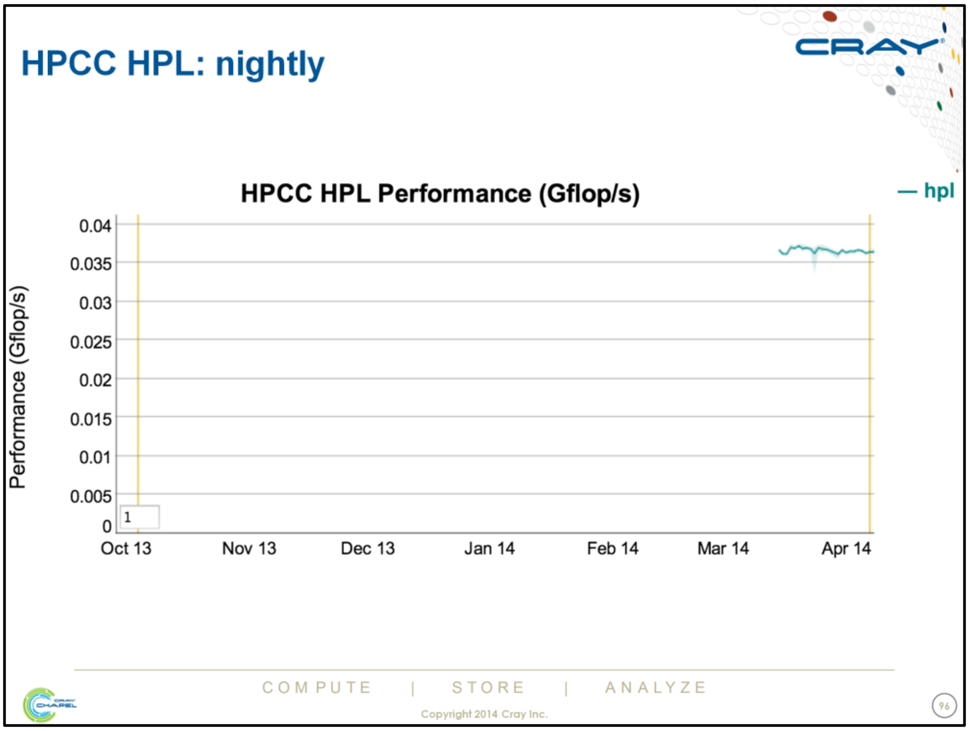

Note that on this slide, higher is better




Note that on this slide, higher is better



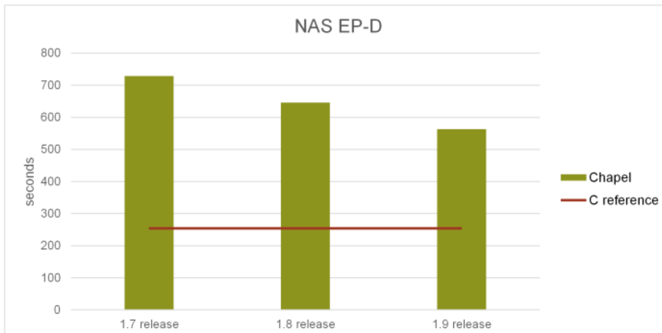

Note that on this slide, higher is better



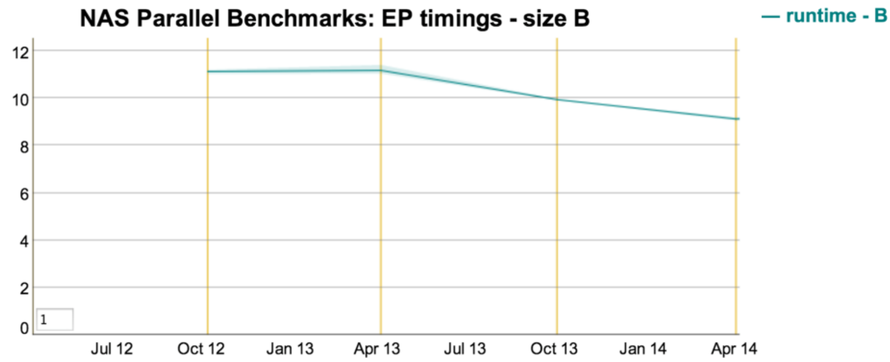

Note that on this slide, higher is better

# NAS Parallel Benchmarks

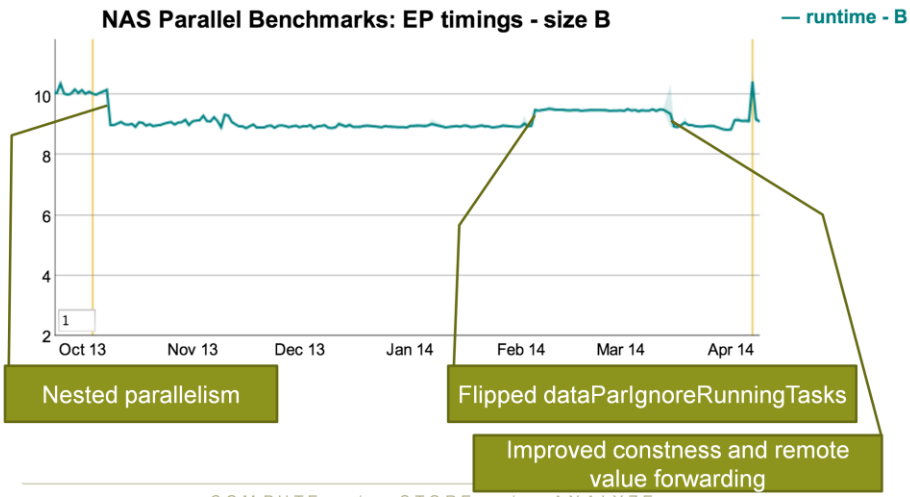
# NAS Parallel EP-D: historical



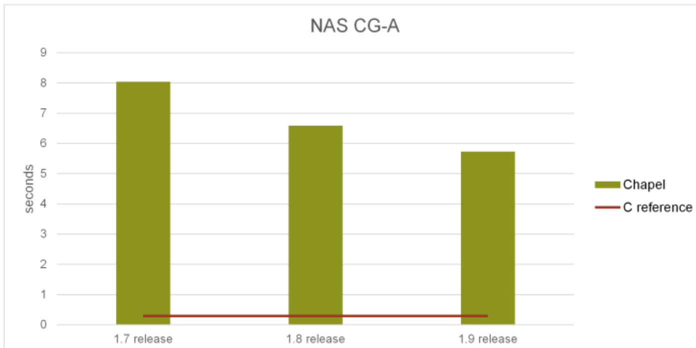
## NAS Parallel EP-B: release-over-release



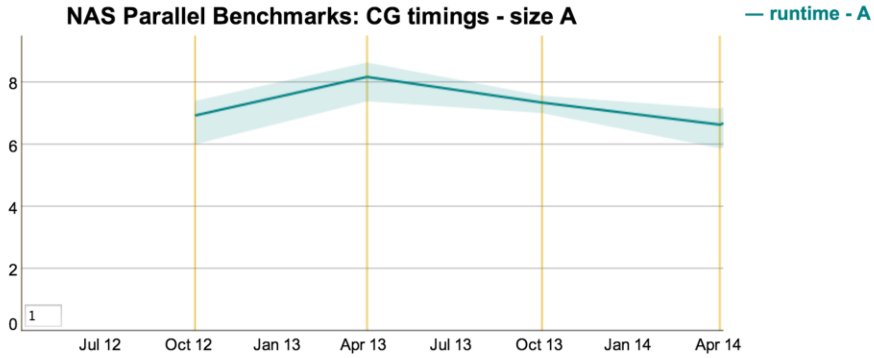
# NAS Parallel EP-B: nightly



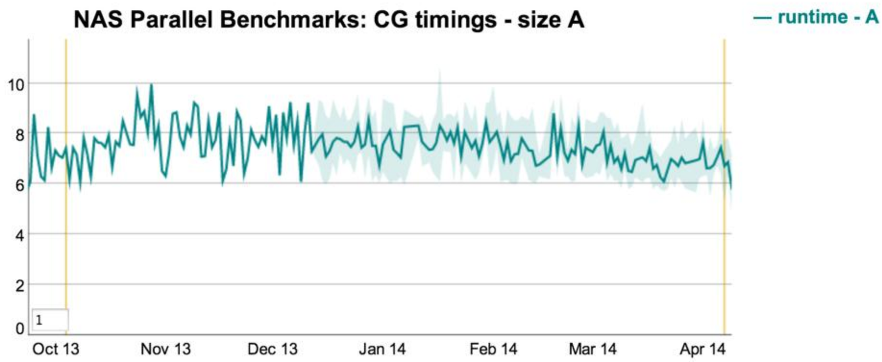
# NAS Parallel CG-A: historical



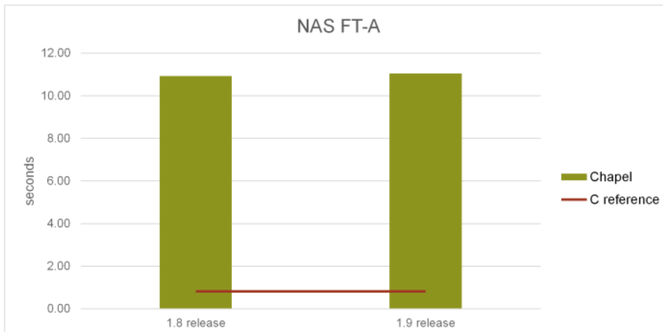
# NAS Parallel CG-A: release-over-release



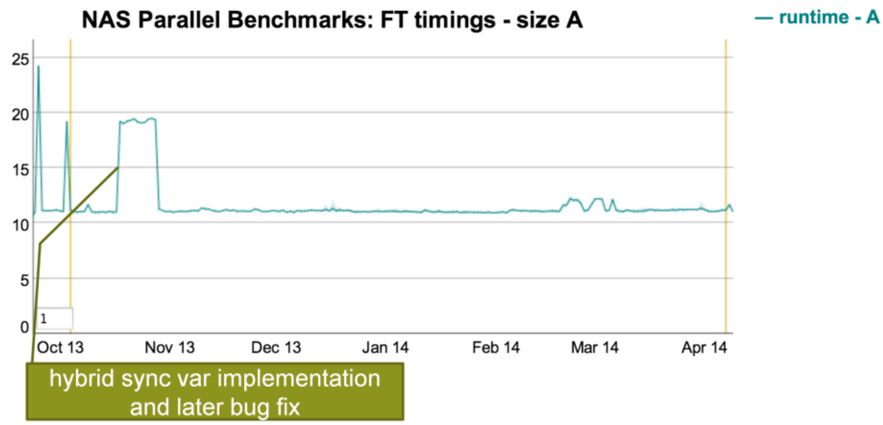
# NAS Parallel CG-A: nightly



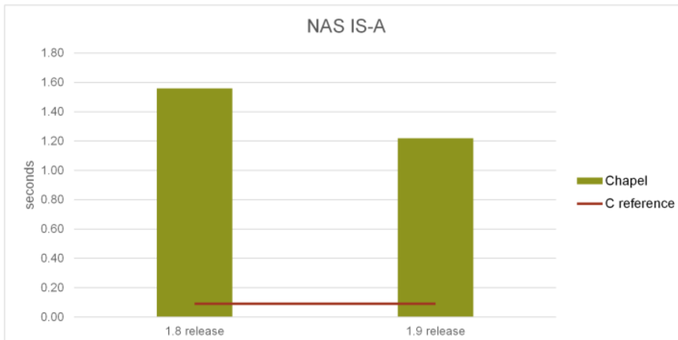
# NAS Parallel FT-A: historical



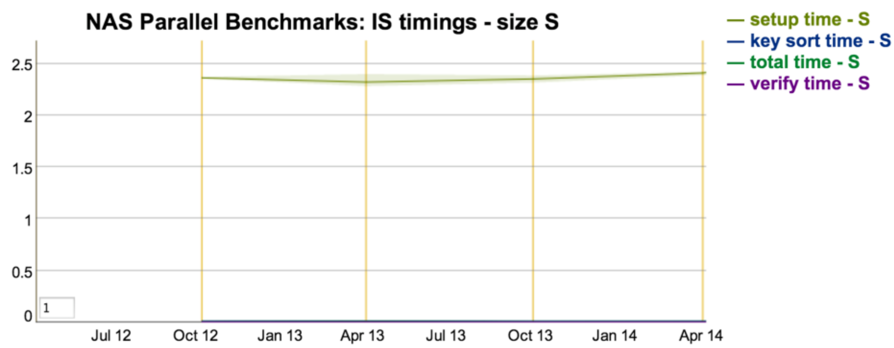
# NAS Parallel FT-A: nightly



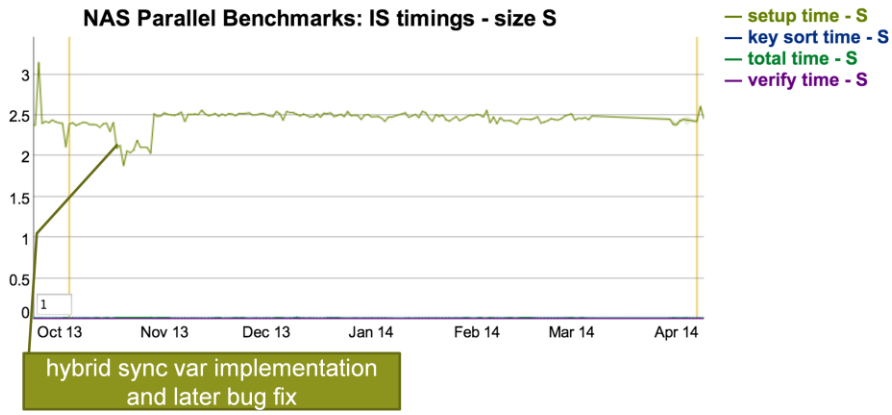
# NAS Parallel IS-A: historical



# NAS Parallel IS-S: release-over-release



# NAS Parallel IS-S: nightly



# Legal Disclaimer

CRAY CHAPEL

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/