Benchmarks and Performance Optimizations

Chapel Team, Cray Inc.

Chapel version 1.18

September 20, 2018

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

- **ugni Improvements**
  - ISx Background
  - Block Transfer Engine (BTE)
  - Active Message (AM) improvements
- **Communication Optimizations**
  - locale.id Communication
  - Barrier Optimizations
- **Qthreads Improvements**
  - Sync Variable Serialization
  - Parallel I/O Improvements
  - Other Sync Variable Improvements
- **Bale Case Study**
  - Histogram Mini-App
    - Background
    - Faster Blocking Atomics
    - Buffered Atomics
- **Memory Leak Improvements**

ugni Improvements

ISx Background

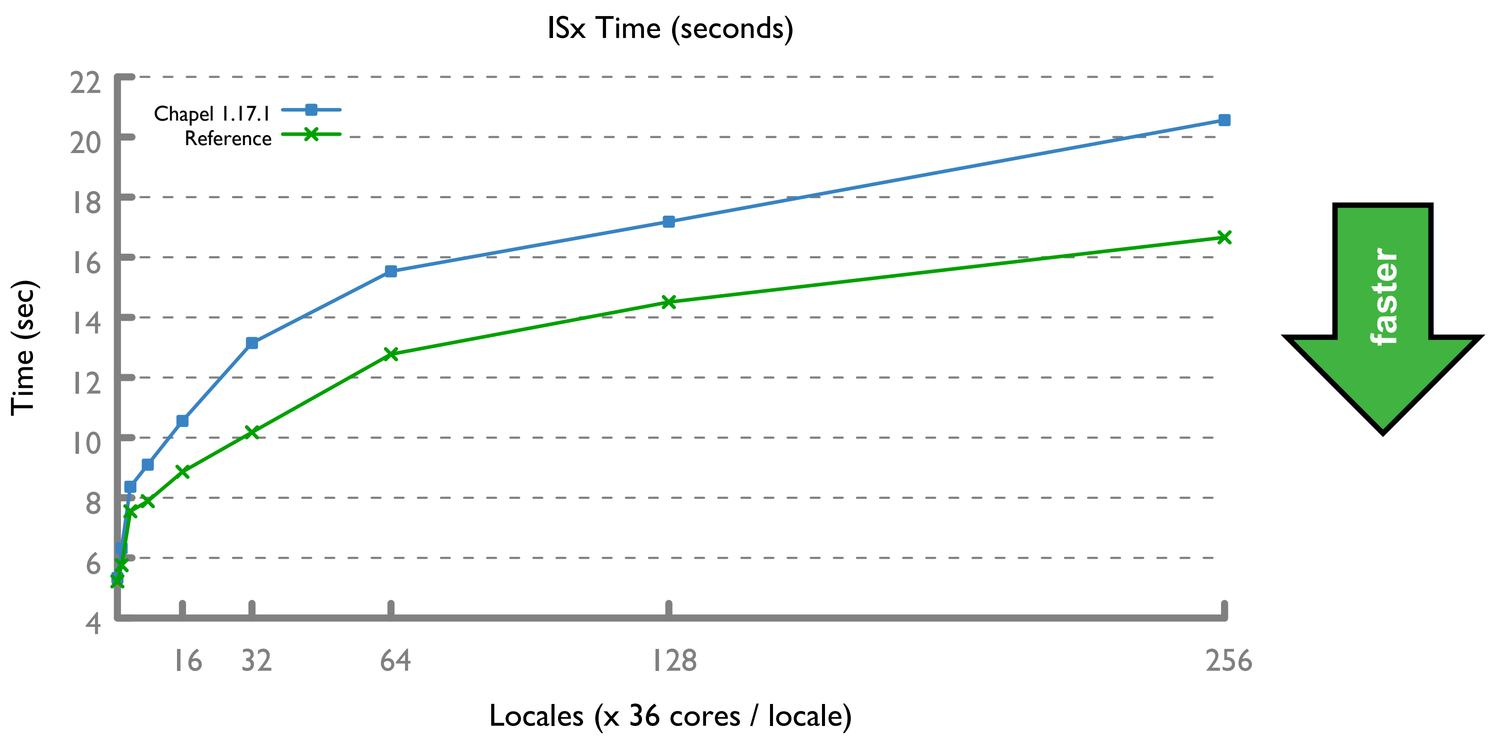
ISx: Background

- **Scalable Integer Sort benchmark**
  - Developed at Intel, published at PGAS 2015
  - SPMD-style computation with barriers
  - Punctuated by all-to-all bucket-exchange pattern
    - buckets being exchanged are relatively large (100's of MBs)
  - References implemented in SHMEM and MPI

- **Chapel implementation introduced in 1.13 release**
  - Motivation: bucket-exchange is a common distributed pattern
  - Benchmark has led to several previous optimizations
    - fast/scalable slicing, bulk transfer optimizations, barrier improvements, …

ISx: Background

- **ISx performance still lagged behind reference SHMEM**
  - Chapel scaled well, but raw performance was up to ~30% behind



ugni: Block Transfer Engine (BTE)

BTE: Background and This Effort

Background: comm=ugni only used Fast Memory Access (FMA)

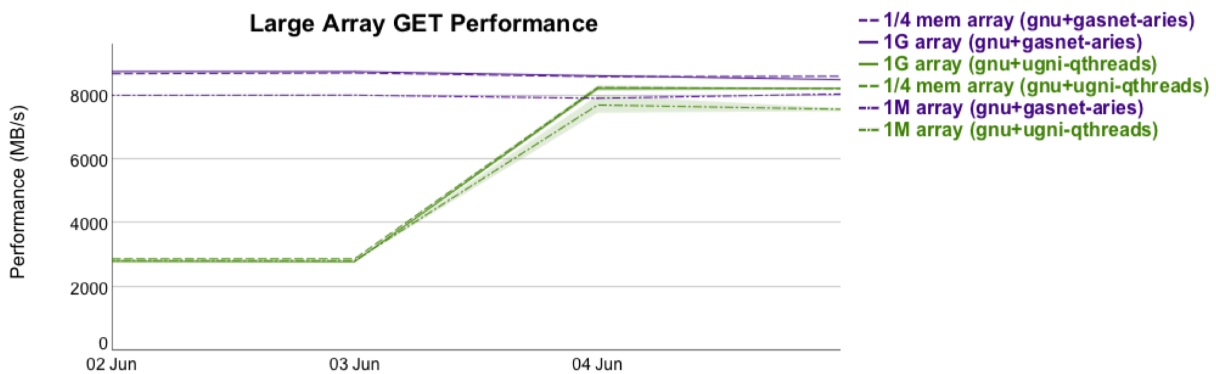
- FMA is optimized for small transfers
- uGNI library also supports Remote Direct Memory Access (RDMA)
  - RDMA is initiated through the Block Transfer Engine (BTE)
  - BTE is optimized for large transfers

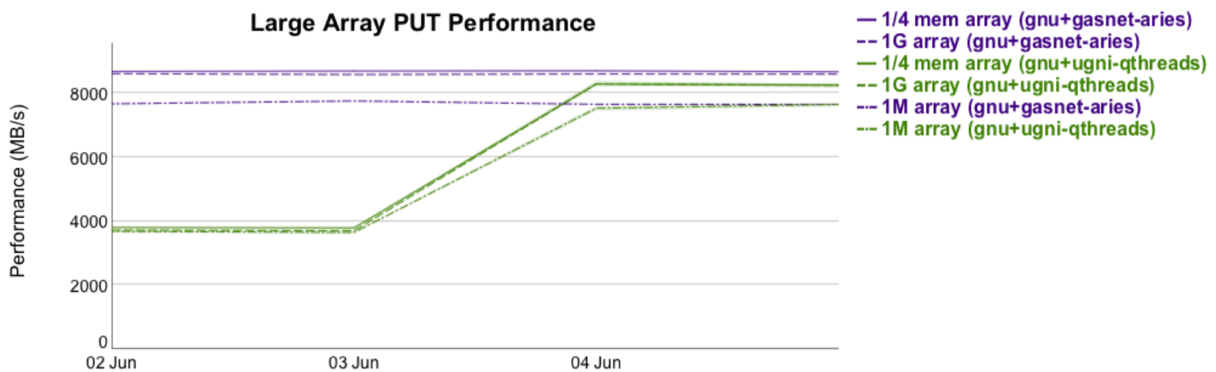
This Effort: Use BTE for PUTs/GETs larger than 4KB

- This significantly increases sustained bandwidth for larger transfers
- 4KB threshold chosen based on tuning, and matches GASNet

BTE: Impact

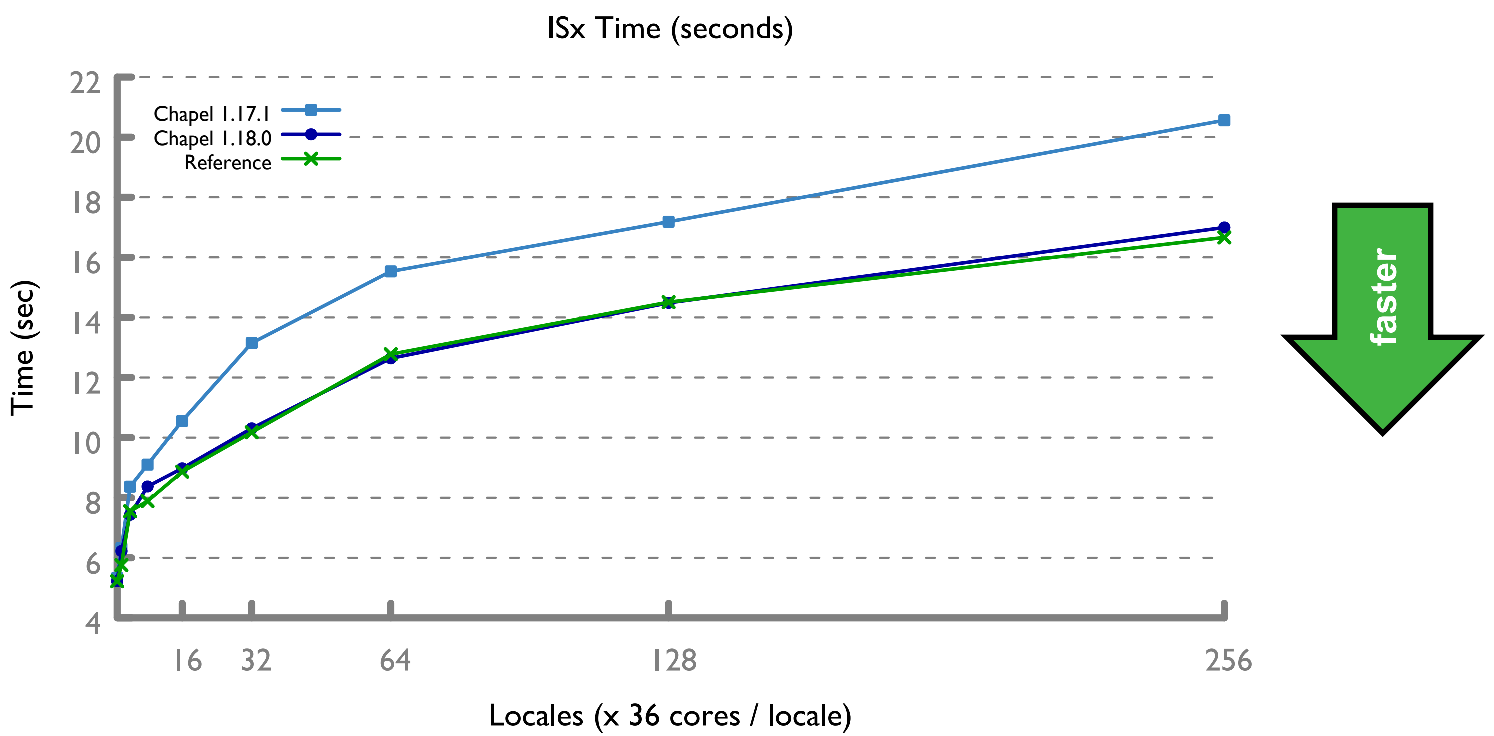
- **Significantly increased sustained transfer bandwidth**
  - Transfers larger than 1MB can sustain max hardware injection rate
    - on par with gasnet-aries, which already used BTE for large transfers





BTE: ISx Impact

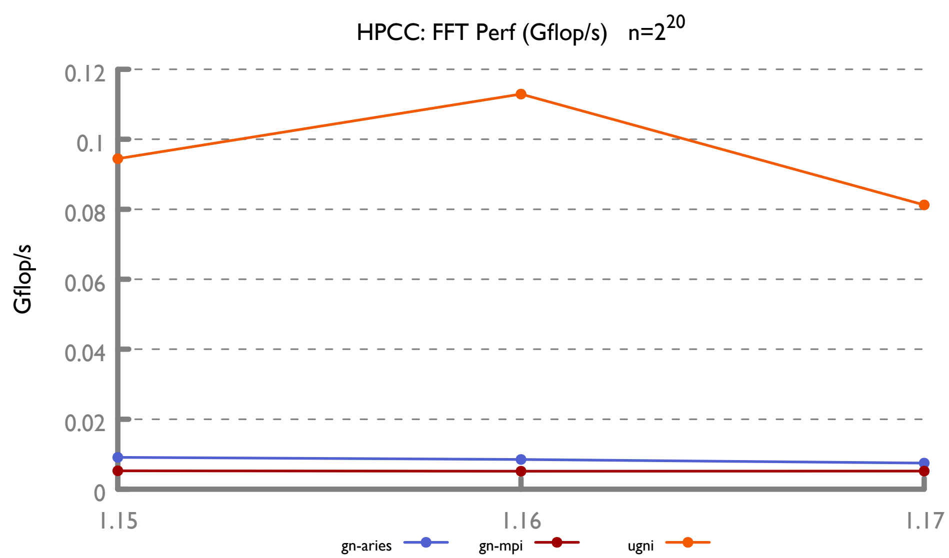
- **ISx performance now on par with reference**
  - No known next steps



ugni: Active Message (AM) Improvements

AM Improvements: Background

- **FFT regressions in 1.17 from “AM done” indicator change**
  - AM done indicators are used to track whether an AM has completed
  - Changed from stack-allocated to heap-allocated pool
    - stack-allocated: cheap allocation, but requires memory registration lookup
    - heap-allocated: contended allocation, but no registration lookup required



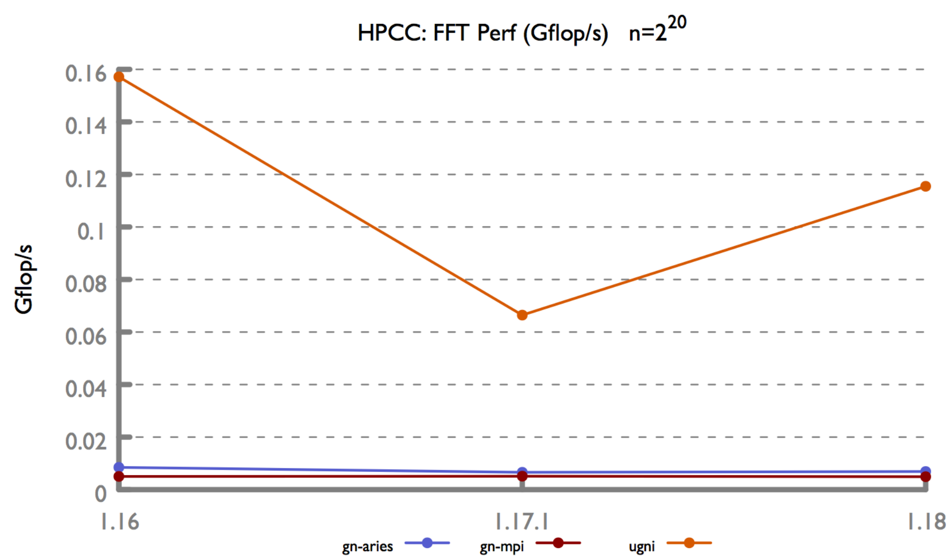
AM Improvements: This Effort and Impact

This Effort: Revert to stack-allocated AM done indicators

- Allocation contention outweighs registration lookup cost

Impact: FFT performance is better, though still behind 1.16

- Remaining hit is from switch to blocking progress thread in 1.17.1
  - needed to mitigate performance hit from Spectre/Meltdown patches



Communication Optimizations

locale.id Communication

locale.id

Background: .id method on a locale returns the locale number

- Useful for data structures reasoning about locality

```
// Suppose A is block distributed and we want to aggregate updates to it.
for indexToUpdate in 1..1000 {
  const dstLocale = A.domain.dist.idxToLocale(indexToUpdate);
  addUpdate(dstLocale.id, indexToUpdate);
}
```

- However dstLocale.id was causing unnecessary communication

This Effort: Removed the unnecessary communication

- Fix suggested by Louis Jenkins

Impact: Surprising source of communication eliminated

- above example now has 0 GETs instead of thousands
- enables progress on prototype aggregation library

Barrier Optimizations

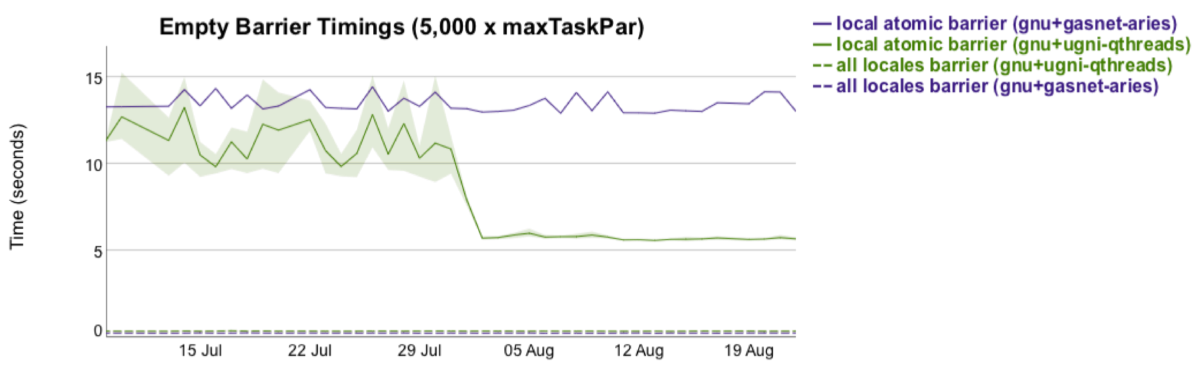
Barrier Optimizations

Background: Barrier implementation is not very scalable

- Scalable `allLocalesBarrier` added in 1.17
  - but the more flexible and default barrier has not been tuned for scale

This Effort: Optimize barriers under network atomics

Impact: Performance improvements for network atomic barrier



Next Steps: Continue to tune default barrier

Qthreads Improvements

Qthreads: Sync Variable Serialization

Sync Var: Background

- **Users ran into perf bottlenecks using sync vars as locks**
  - Example from “Parallel Sparse Tensor Decomposition in Chapel”
    - Presented by Thomas Rolinger at CHIUW 2018

2.) Porting SPLATT to Chapel:

Mutex Pool

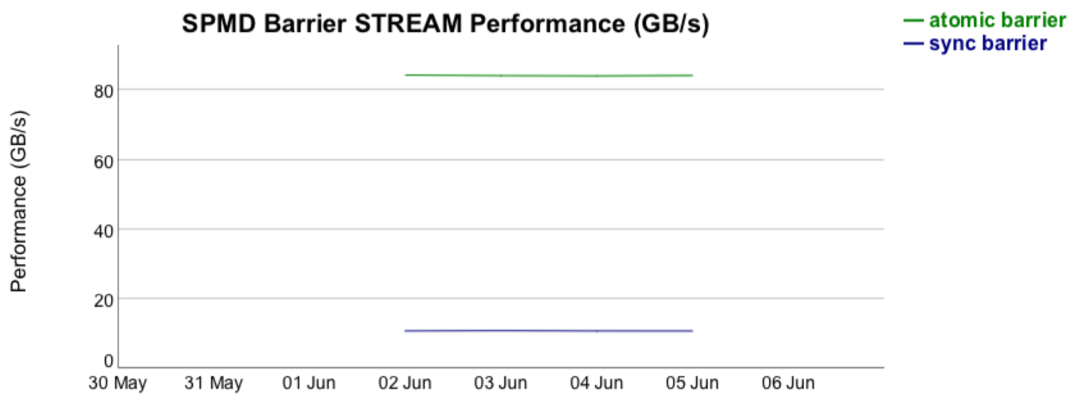
- SPLATT uses a mutex pool for some of the parallel MTTKRP routines to synchronize access to matrix rows
- Chapel currently does not have a native lock/mutex module
  - Can recreate behavior with **sync** or **atomic** variables
  - We originally used **sync** variables, but later switched to **atomic** (see Performance Evaluation section).

Sync Var: Background

- **Made a simpler benchmark to investigate**
  - SPMD Stream triad that barriers

```
coforall tid in 0..#numTasks {
  barrier.barrier();
  for i in chunk(1..m, numTasks, tid) do
    A[i] = B[i] + alpha * C[i];
}
```

- **Discovered that sync-based barrier serialized execution**



Sync Var: Background and This Effort

Background: Qthread syncs optimized for producer/consumer

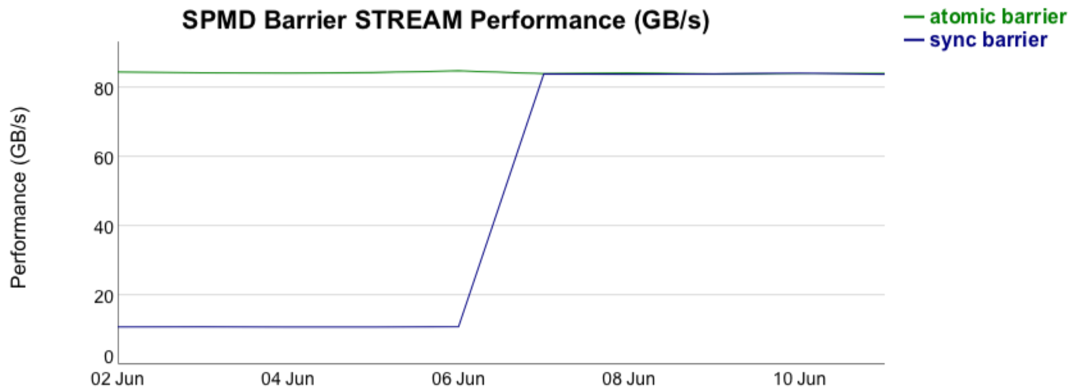
- Unblocked sync vars scheduled tasks onto the current thread
  - assumed producer would block, and consumer could reuse data in cache
- This is not ideal for sync vars used as locks/barriers
  - serialized all tasks onto the same thread

This Effort: Reschedule woken task onto the original thread

- Avoids task serialization, but can hurt producer/consumer perf
  - opened issue with Qthreads team, pursuing better options
  - in the meantime our workaround is better overall for Chapel

Sync Var: Impact

- **Sync variables no longer serialize execution**
  - Sync-based barrier on par with atomic-based barrier for STREAM

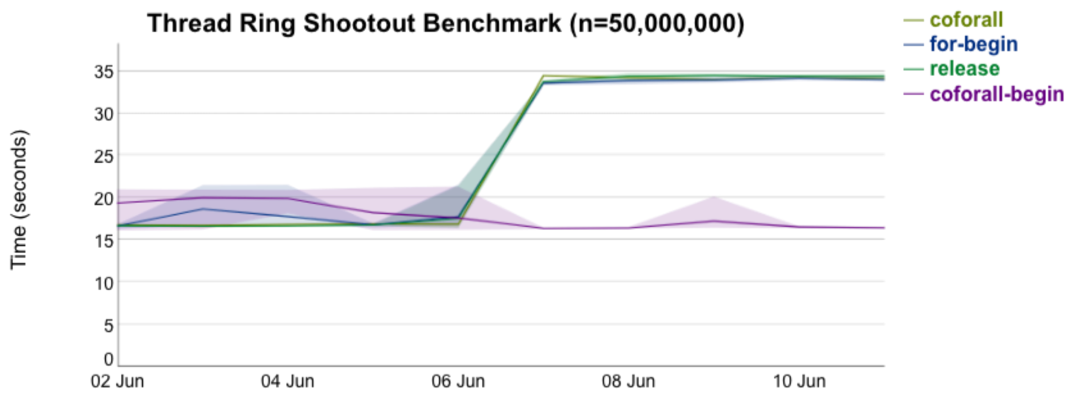


  - SPLATT performance with sync var locks is much better

Config	Time
1.17.1 Sync Locks	19.1s
1.18.0 Sync Locks	5.6s
Atomic Locks	5.4s

Sync Var: Negative Impact

- **Caused a performance regression for threadring**
  - Unfairly benefitted from previous serialization
    - not a code we are deeply invested in



Qthreads: Parallel I/O Improvements

Parallel I/O: Background

- **Saw serious performance degradation with parallel I/O**
  - Especially when 2 Chapel executables ran concurrently on a node

```
coforall t in 1..here.maxTaskPar {
  for i in 1..100 do
    writeln(t, ": " , i);
}
```

```
Starting first instance of 'time -p ./io-slowdown'
0.12s
0.09s
0.30s
0.07s
Starting second concurrent instance of 'time -p ./io-slowdown'
7.97s
        7.97s
13.68s
        13.68s
```

Output from 2nd instance

Output from 1st instance

Parallel I/O: This Effort and Impact

This Effort: Transitioned from spinlock to sync var lock

- Enabled by sync var serialization fixes

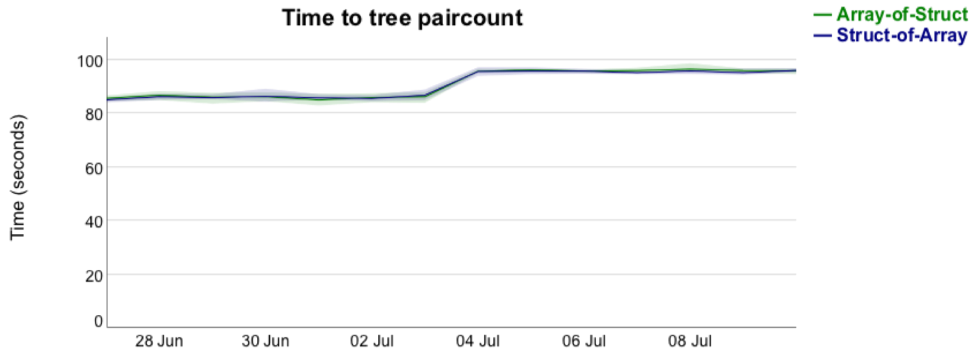
Impact: Improved parallel I/O performance

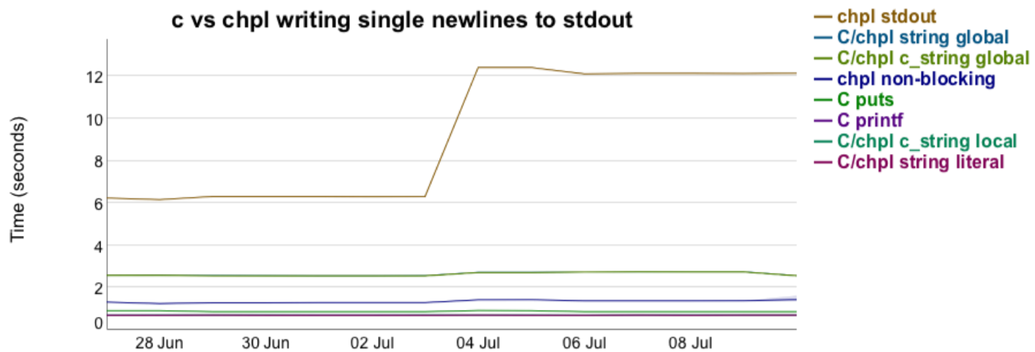
- Especially for concurrent runs

```
Starting first instance of 'time -p ./io-slowdown'
0.07s            (~0.12s previously)
0.07s
0.06s
0.03s
Starting second concurrent instance of 'time -p ./io-slowdown'
        0.27s  (~10.0s previously)
0.28s
        0.18s
0.18s
0.35s
```

Parallel I/O: Negative Impact

- **Serial I/O performance suffered**
  - For uncontested access, an atomic lock is faster than a sync lock
    - believe parallel I/O improvements outweigh these regressions
    - advanced users can manually disable locking for serial I/O





Parallel I/O: Next Steps

- **Transition to a hybrid lock**
  - Use an atomic for uncontested access, fall back to sync if contested
- **Investigate compiler optimizations**
  - May be able to eliminate locking when access is provably serial

Qthreads: Other Sync Var Improvements

Sync Improvements: Background and Effort

Background: Qthreads has 2 sync variable implementations

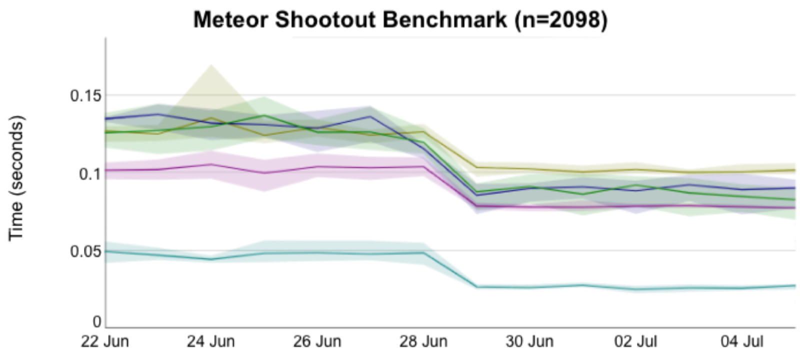
- aligned_t – Full/Empty Bit state stored externally, 64 bits available
  - chapel sync vars map to this type (since we need to store 64-bit types)
- syncvar_t – 3 bits to store Full/Empty Bit state, leaving 61 bits for data
  - was used in runtime shim in a few places

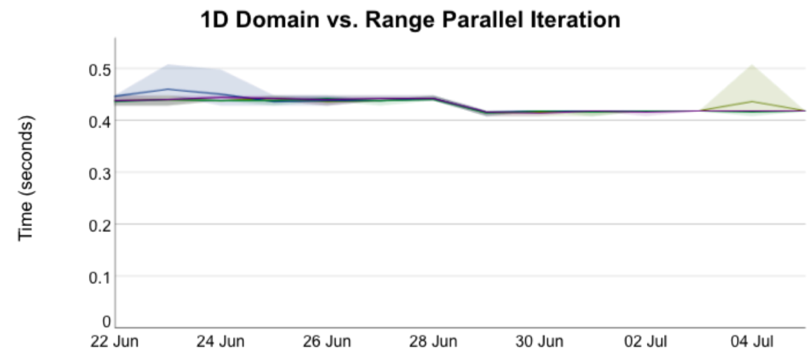
This Effort: Change runtime shim uses of syncvar_t to aligned_t

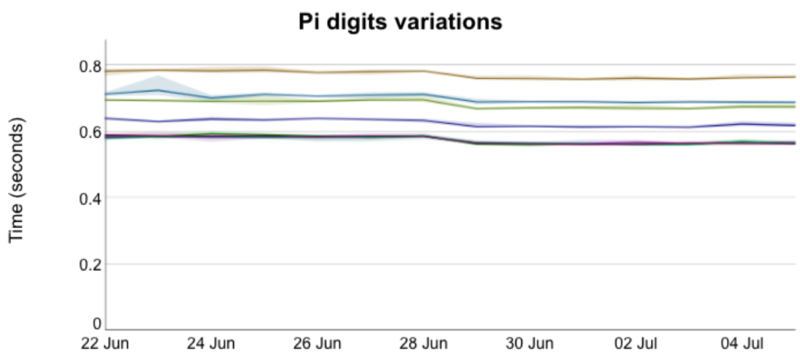
- syncvar_t still has serialization issue (only fixed for aligned_t)
- aligned_t version is better tested (since Chapel types map to it)

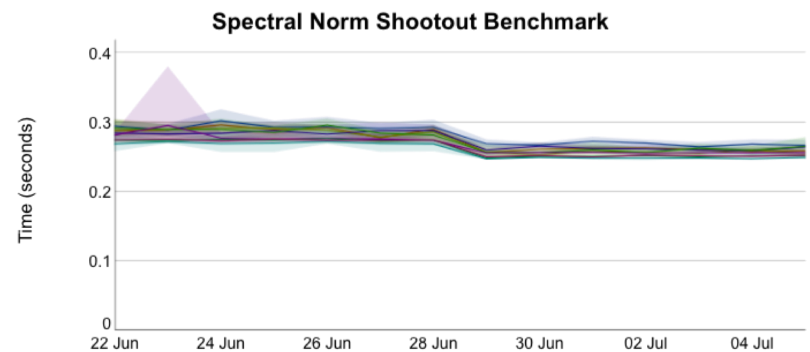
Sync Improvements: Impact

- Performance improvements for several benchmarks









Bale Case Study

Bale: Background

- **Bale is a collection of mini-applications in UPC/SHMEM**
  - Tests various communication idioms and patterns
    - Histogram (stresses network atomics)
    - Indexgather (stresses remote GETs)
    - Toposort
- **Bale also contains aggregated communication libraries**
  - Compares elegant/intuitive code vs. more complex aggregated code
  - For our initial study, we focused on performance of elegant versions
    - implemented versions of histogram, indexgather, and toposort
    - started tuning performance of histogram first

Bale Histogram Background

Histogram: Background

- **Histogram randomly updates an array of network atomics**
  - Idiom is similar to our atomic-based version of RandomAccess (RA)

Default UPC

```
for(i = 0; i < T; i++) {
  counts[index[i]] += 1;
}
```

Default Chapel

```
forall r in rindex  {
  A[r].add(1);
}
```

Optimized UPC

```
for(i = 0; i < T; i++) {
  #pragma pgas defer_sync
  counts[index[i]] += 1;
}
lgp_barrier();
```

Histogram: Background

- **By default, network operations are "blocking"**
  - Have to wait for an acknowledgement (ACK) from remote locales
  - Required by Memory Consistency Model (MCM)
    - "sequential consistency for data-race-free programs"

```
var a: atomic int;
on Locales[1] {
  a.add(1);
  writeln(a.read()); // must print 1
}
```

- **Blocking operations limit network injection rate**
  - Have to wait for round-trip network ACK
    - instead of issuing multiple operations back-to-back

Histogram: Background

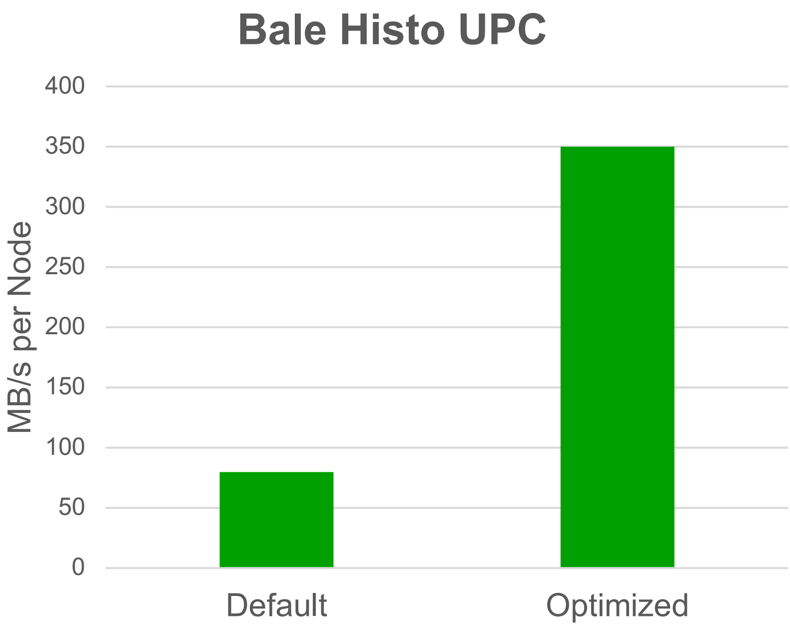
- **Cray UPC/SHMEM can drop to more relaxed MCM modes**
  - “Use the ‘pgas defer_sync’ directive to force all references in the next statement to be non-blocking”

Default UPC

```
for(i = 0; i < T; i++) {
  counts[index[i]] += 1;
}
```

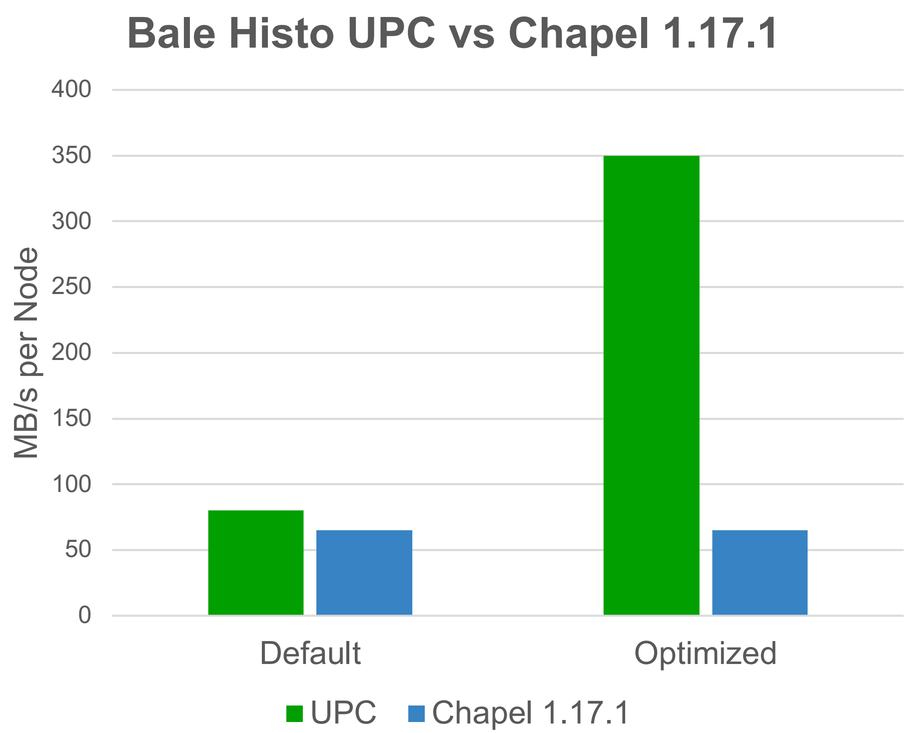
Optimized UPC

```
for(i = 0; i < T; i++) {
  #pragma pgas defer_sync
  counts[index[i]] += 1;
}
lgp_barrier();
```



Histogram: Background

- **Chapel performance was ~15% behind default**
  - And ~5.5x off from the optimized variant



Faster Blocking Atomics

Faster Atomics: Background

- **Used to yield continuously while waiting for remote ACK**
  - Yielding allows for comm/compute overlap
  - Discovered that task-yield is more expensive than expected
    - tasks often in middle of yield when ACK comes in

```
cdi = post_fma(locale, post_desc)          // initiate transaction (post to NIC)

do {
  chpl_task_yield();                       // yield every iter

  consume_all_outstanding_cq_events(cdi);
} while (!atomic_load_bool(&post_done));   // blocking wait for transaction to complete
```

Faster Atomics: This Effort

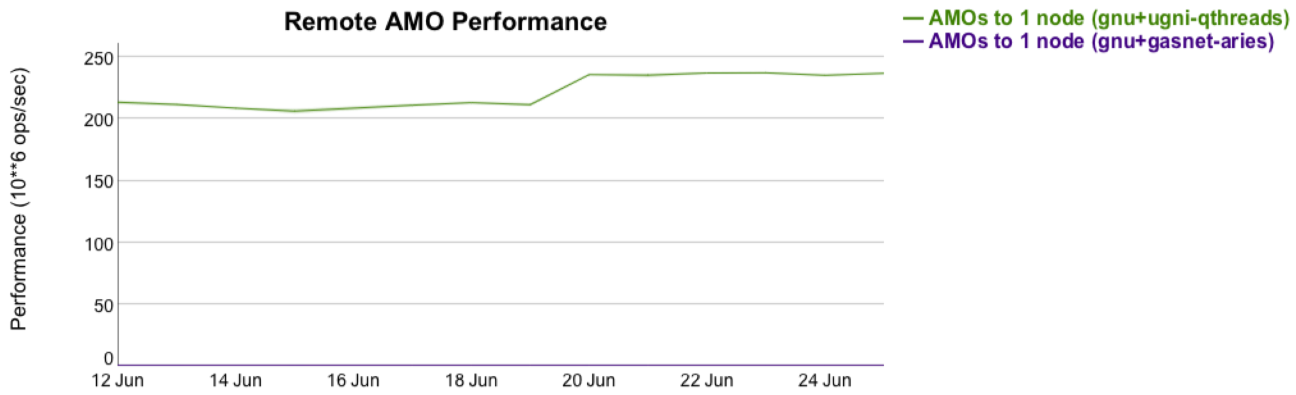
- **Switch to yielding initially, then every 64 tries**
  - Still allows for comm/compute overlap when numTasks > numCores
    - when not oversubscribed, can process ACK sooner
  - Value chosen experimentally, 32 and 128 also worked well
    - chose middle ground, longer-term solution is to optimize task-yields

```
cdi = post_fma(locale, post_desc)                    // initiate transaction (post to NIC)

do {
  if ((iters & 0x3F) == 0) chpl_task_yield();       // yield initially, then 1/64 iters
  iters++;
  consume_all_outstanding_cq_events(cdi);
} while (!atomic_load_bool(&post_done));             // blocking wait for transaction to complete
```

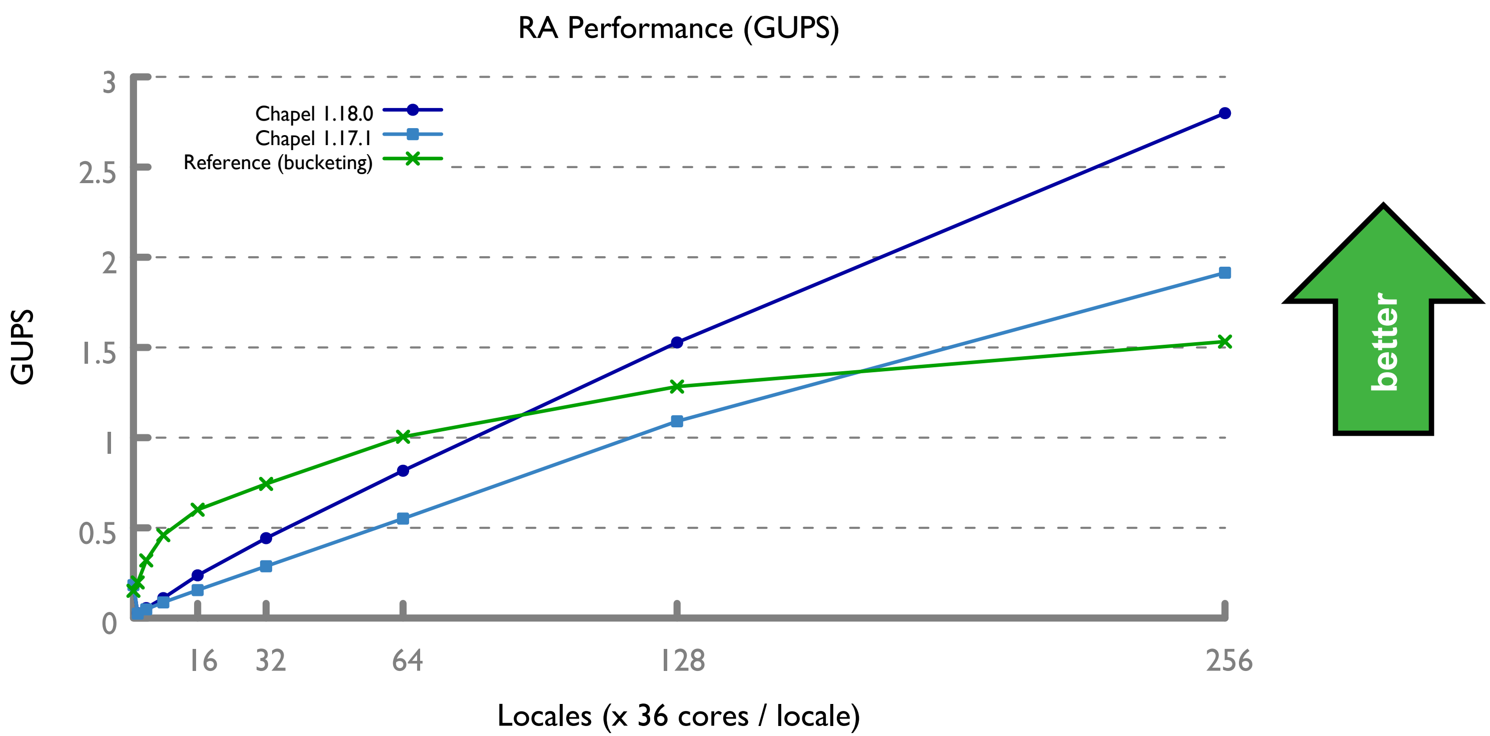
Faster Atomics: Impact

- **Improved blocking atomic performance**
  - Better performance for many-to-one atomic microbenchmark



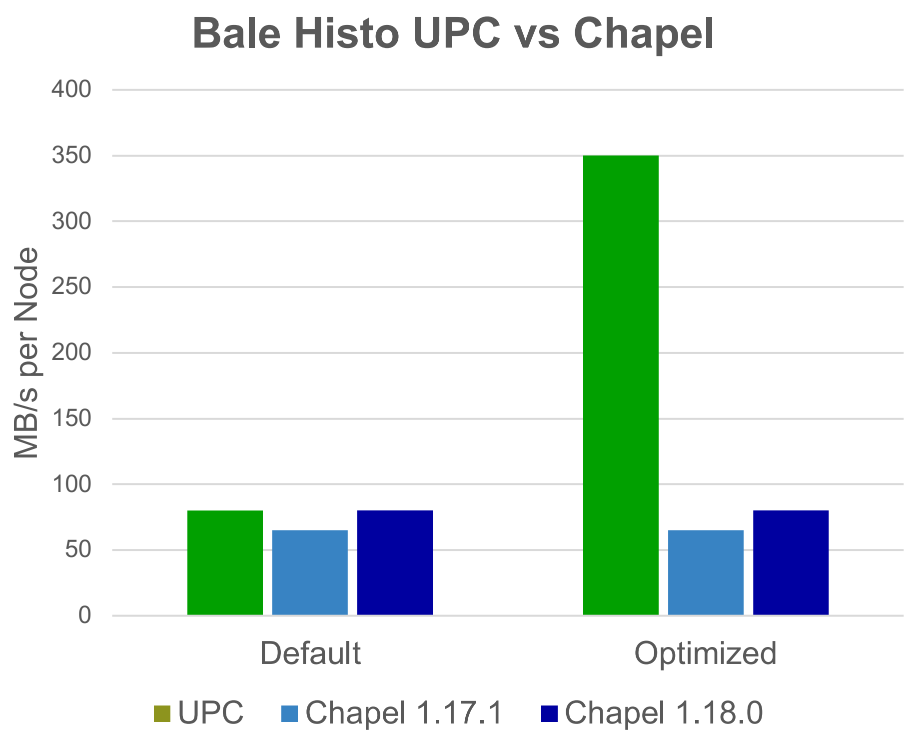
Faster Atomics: Impact

- **Improved blocking atomic performance**
  - Better performance for RA-atomics benchmark



Faster Atomics: Histogram Impact

- **Chapel performance on par with default UPC**
  - Still ~4.5x off from the optimized variant



Buffered Atomics

Buffered Atomics: Background and Effort

Background: Chapel had no way to drop to more relaxed MCM

- Foundation/placeholder in the spec: "Unordered Memory Operations"
  - but no implementation, source of optimized performance gap

This Effort: Added "buffered" atomics to express unordered ops

- Operations are not sequentially consistent, must be explicitly flushed
- Implemented in a package module:
  - https://chapel-lang.org/docs/1.18/modules/packages/BufferedAtomics.html
- Allowed for fast prototype without language/spec changes

```
var a: atomic int;
a.addBuff(1);
writeln(a);           // can print 0 or 1
flushAtomicBuff();
writeln(a);           // must print 1
```

Buffered Atomics: This Effort

- **Wrote a buffered version of histogram:**

Default Chapel

```
forall r in rindex  {
  A[r].add(1);
}
```

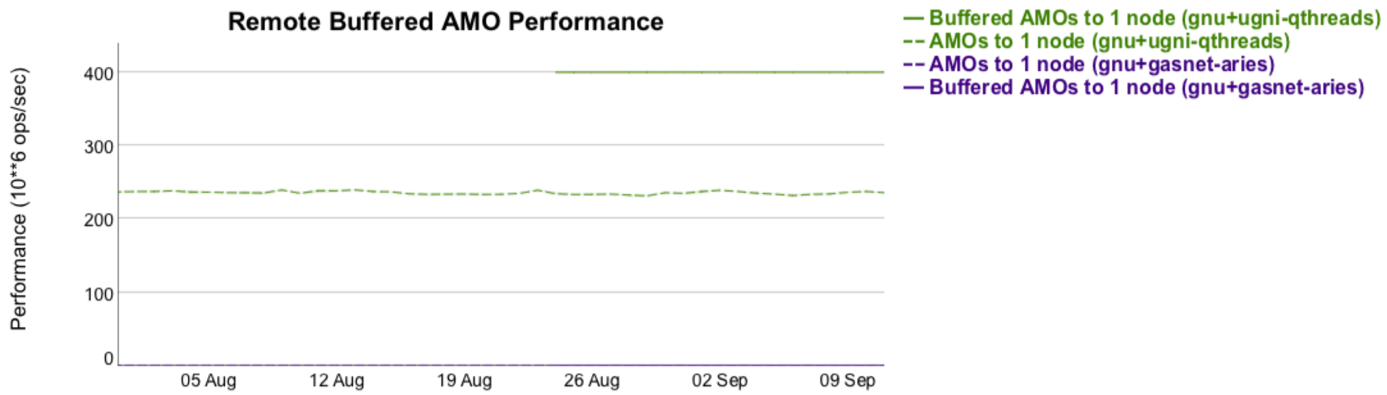
Optimized Chapel

```
forall r in rindex {
  A[r].addBuff(1);
}
flushAtomicBuff();
```

- **Under the hood: operations stored in thread-local buffers**
  - Buffers are flushed when full or on calls to 'flushAtomicBuff()'
  - We initiate transactions all at once with:
    - ugni "chained" transactions for CLE 5.2UP04 and up (up to 5x perf gain)
    - non-blocking transactions for older versions of CLE (up to 2.5x perf gain)

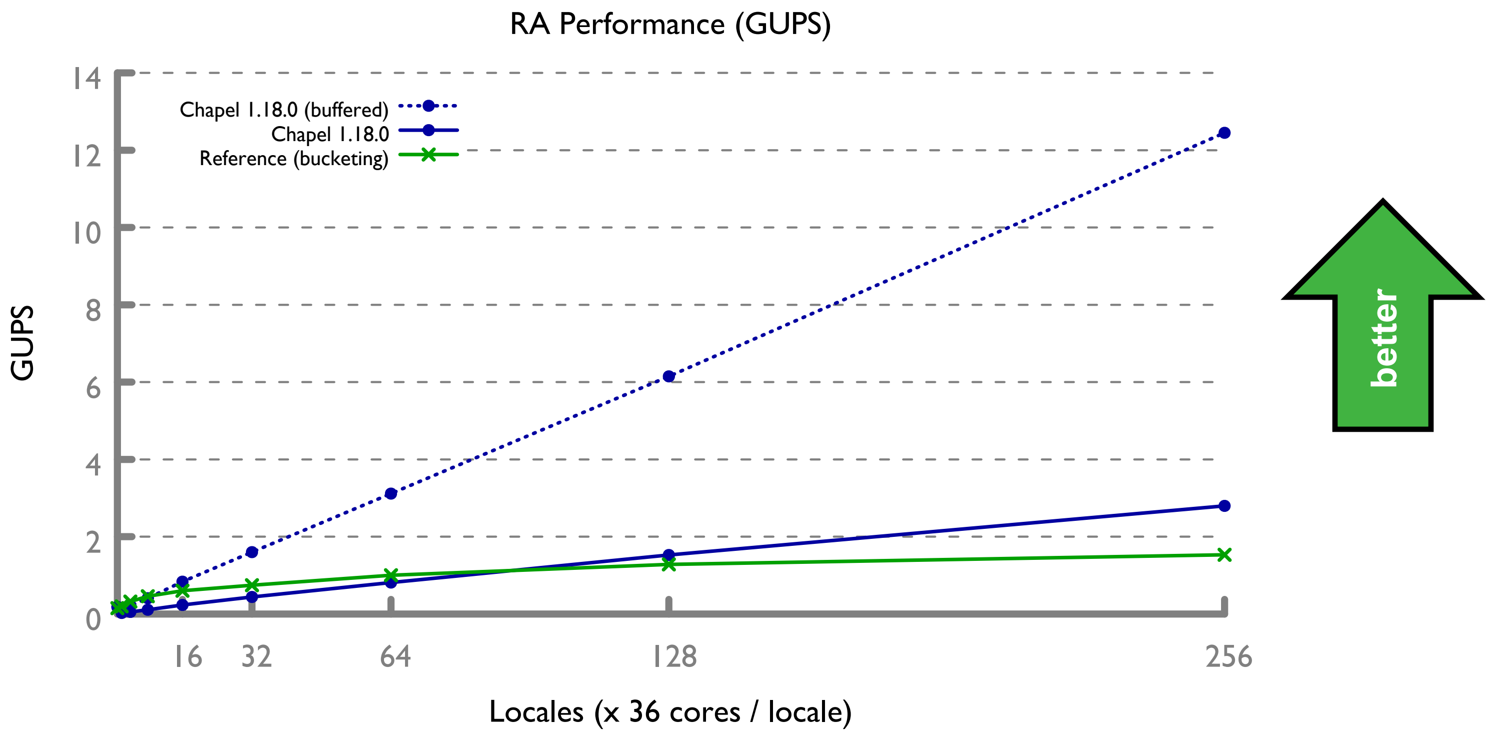
Buffered Atomics: Impact

- **Better performance for codes that can use buffered ops**
  - ~1.5x improvement for many-to-one microbenchmark



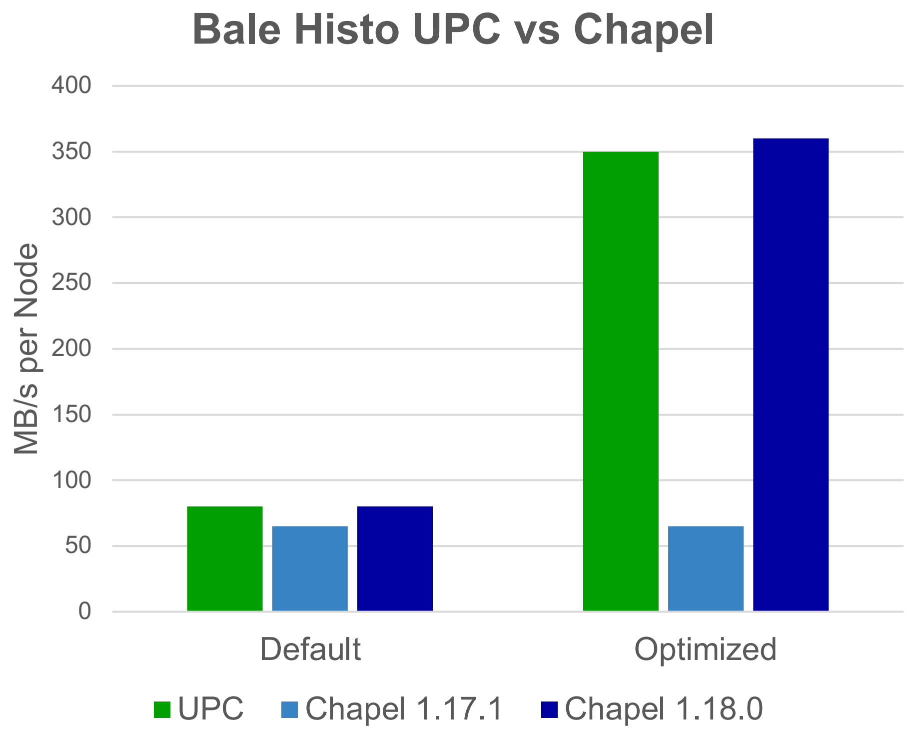
Buffered Atomics: Impact

- **Better performance for codes that can use buffered ops**
  - ~4.5x improvement for buffered RA-atomics benchmark



Buffered Atomics: Histogram Impact

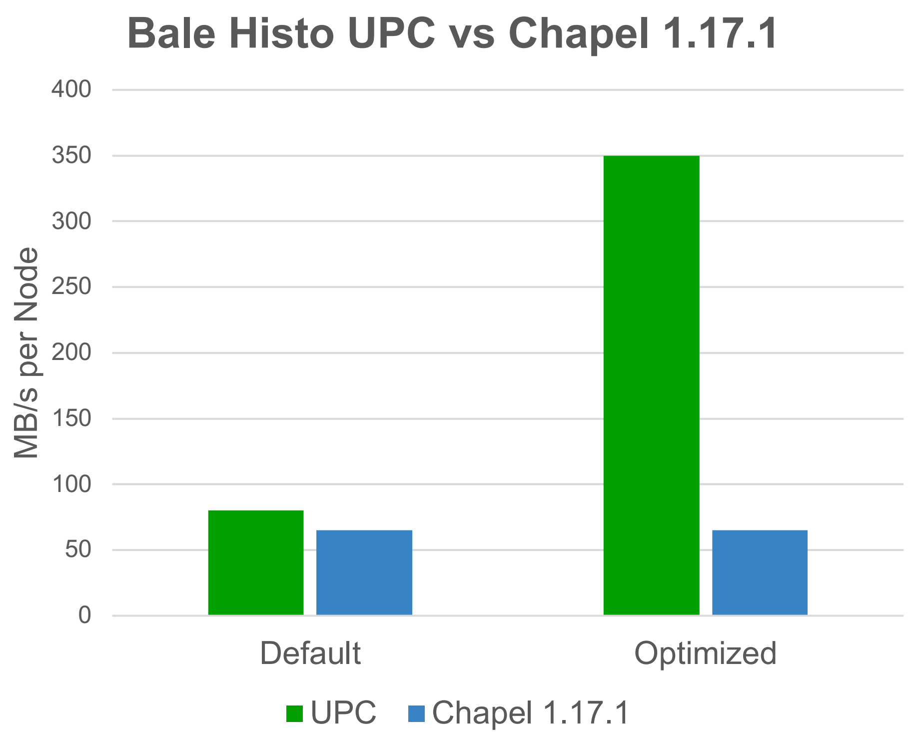
- **Chapel performance on par with default UPC**
  - And for the optimized variant



Bale Histogram Summary

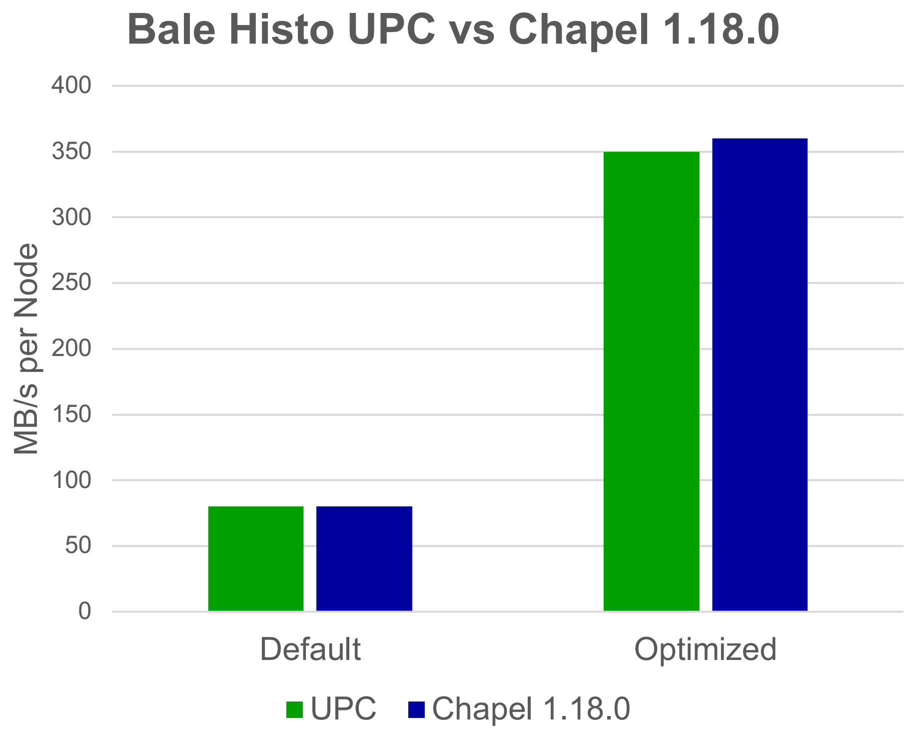
Histogram: Summary

- **In 1.17.1 blocking performance was ~15% behind UPC**
  - Optimized performance was ~5.5x off



Histogram: Summary

- **In 1.18.0 performance is on par with UPC**
  - Result of optimizing blocking atomics and adding buffered atomics



Histogram: Next Steps

- **Improve elegance of optimized histogram code**
  - `addBuff()` reveals too much about the implementation
    - explicit flush is cumbersome

```
forall r in rindex do
  A[r].addBuff(1);
flushAtomicBuff();
```

  - Add a more general syntax for super-relaxed operations
    - Current implementation only supports atomic operations

```
deferSync do forall r in rindex do // 'deferSync' as a proposed syntax
  A[r].add(1);
```

  - Add compiler optimization to automatically perform transformation
    - Not always possible, but cases like this should be straightforward

Bale: Summary and Next Steps

Summary: Ported Bale mini-apps to Chapel

- Optimized histogram to match UPC performance

Next Steps:

- Optimize indexgather and toposort
  - indexgather tuning is already underway
- Improve elegance
  - need a cleaner way to express unordered operations
- Start investigating buffered/aggregated examples
  - aggregation buffers updates to remote locales, permits bulk communication

Memory Leak Improvements

Memory Leaks: Background + This Effort

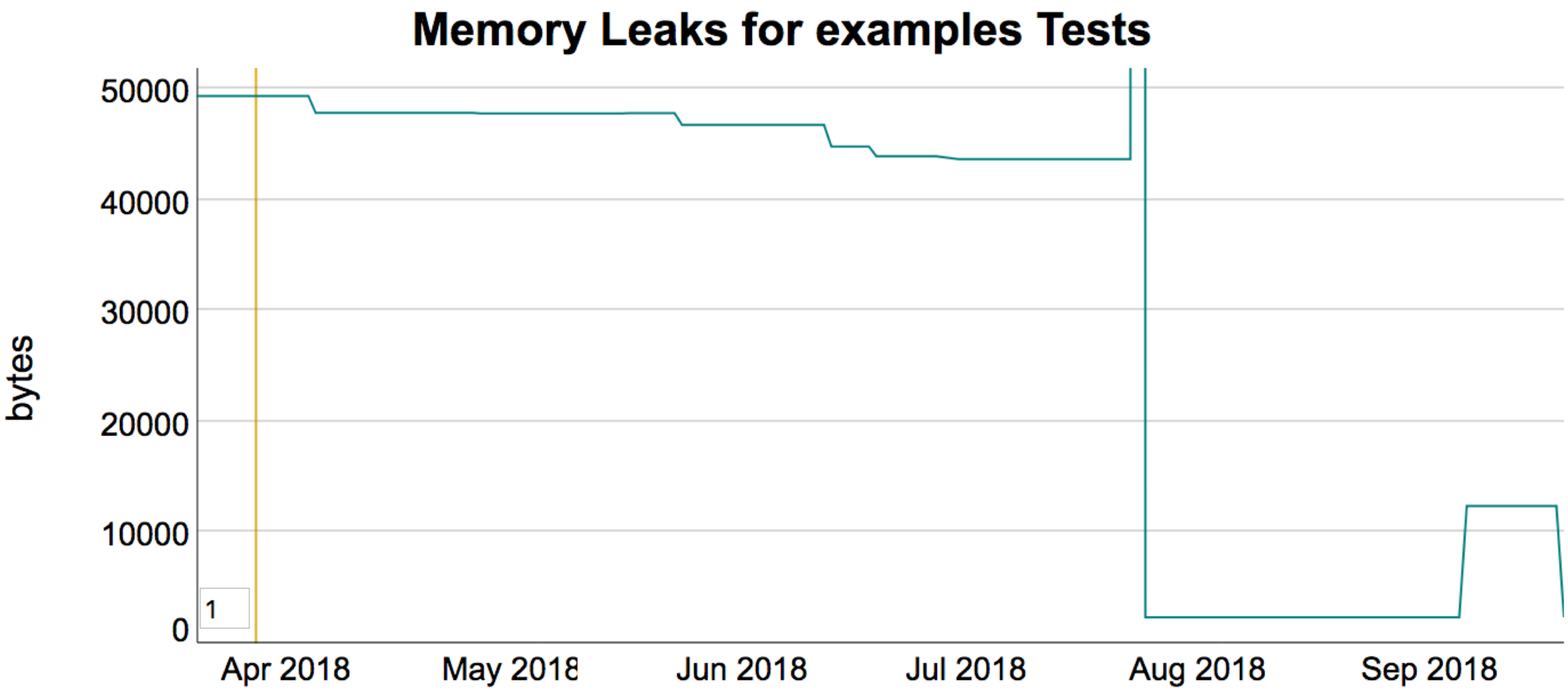
Background:

- Historically, Chapel testing has leaked a large amount of memory
- Chapel 1.15 and 1.16 closed major sources of large-scale leaks
- Chapel 1.17 reduced leaked memory in testing by another 50%

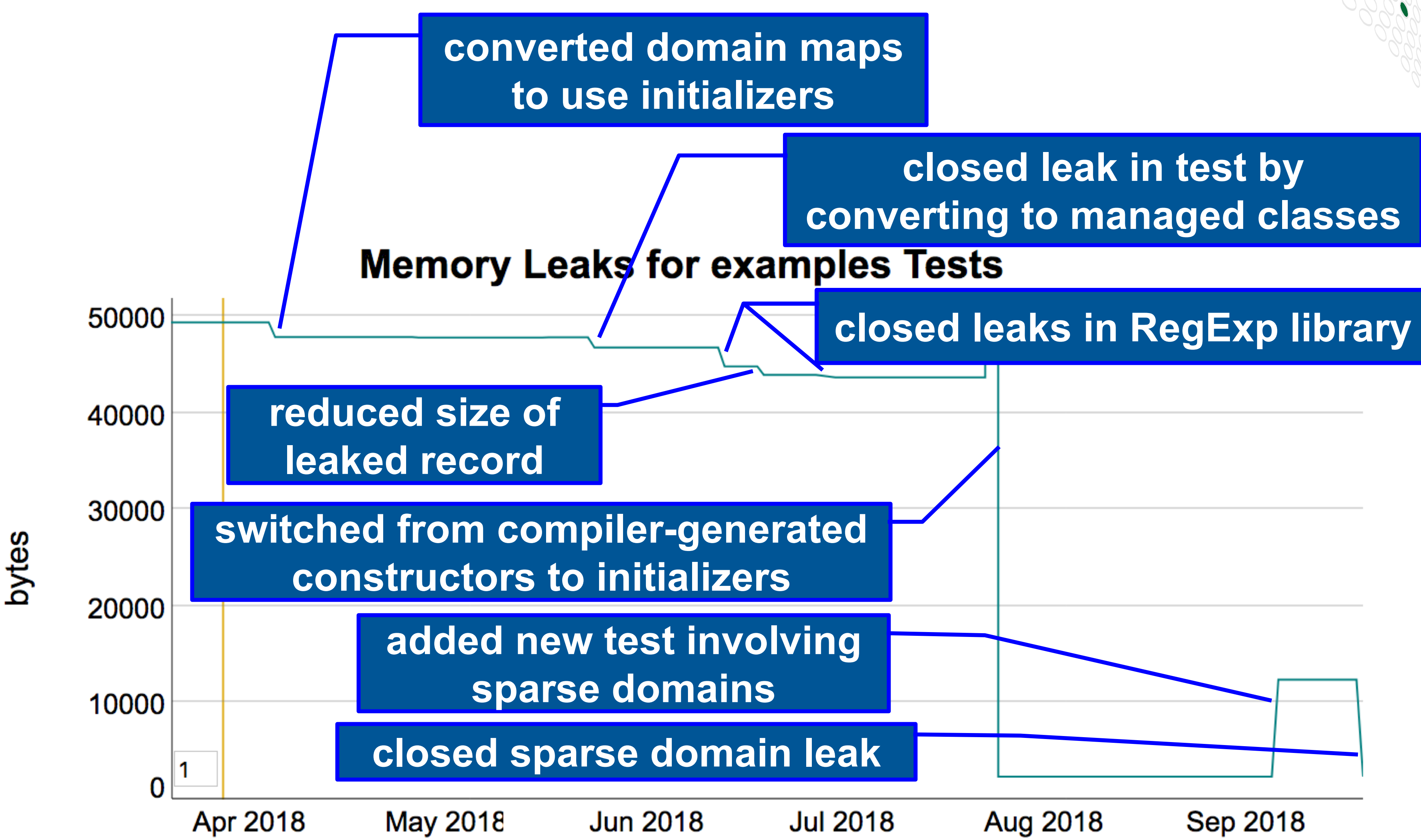
This Effort:

- Closed several classes of leaks reported by nightly testing:
  - leaks caused by using constructors rather than initializers
  - minor leaks in several library modules:
    - RegExp, DateTime, CPtr, List, FileSystem
  - leaks in tests that were fixed when converting to managed class types
- Just after cutting the 1.18 branch, closed a leak in CS sparse domains
  - (reflected in these notes, but not included in the release)

Memory Leaks for Examples in Release

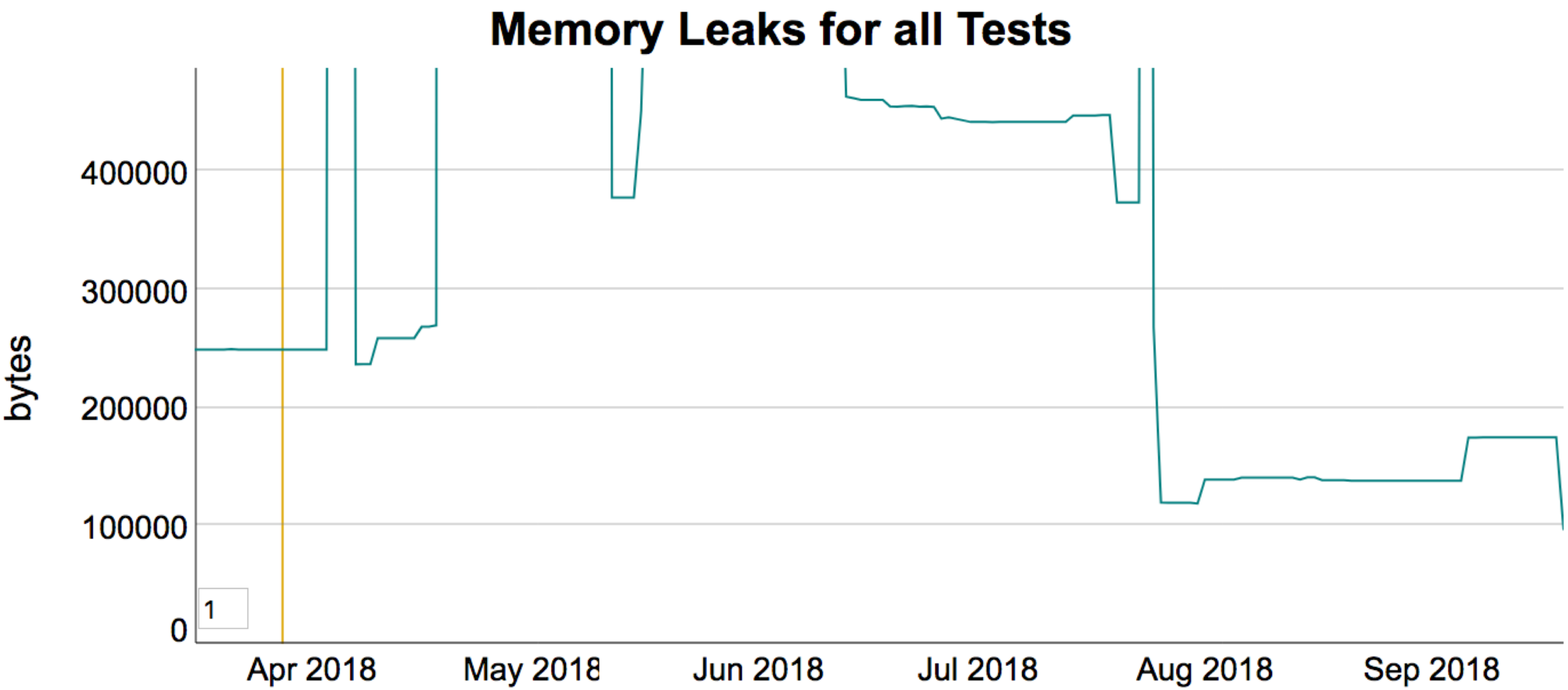


Memory Leaks for Examples in Release

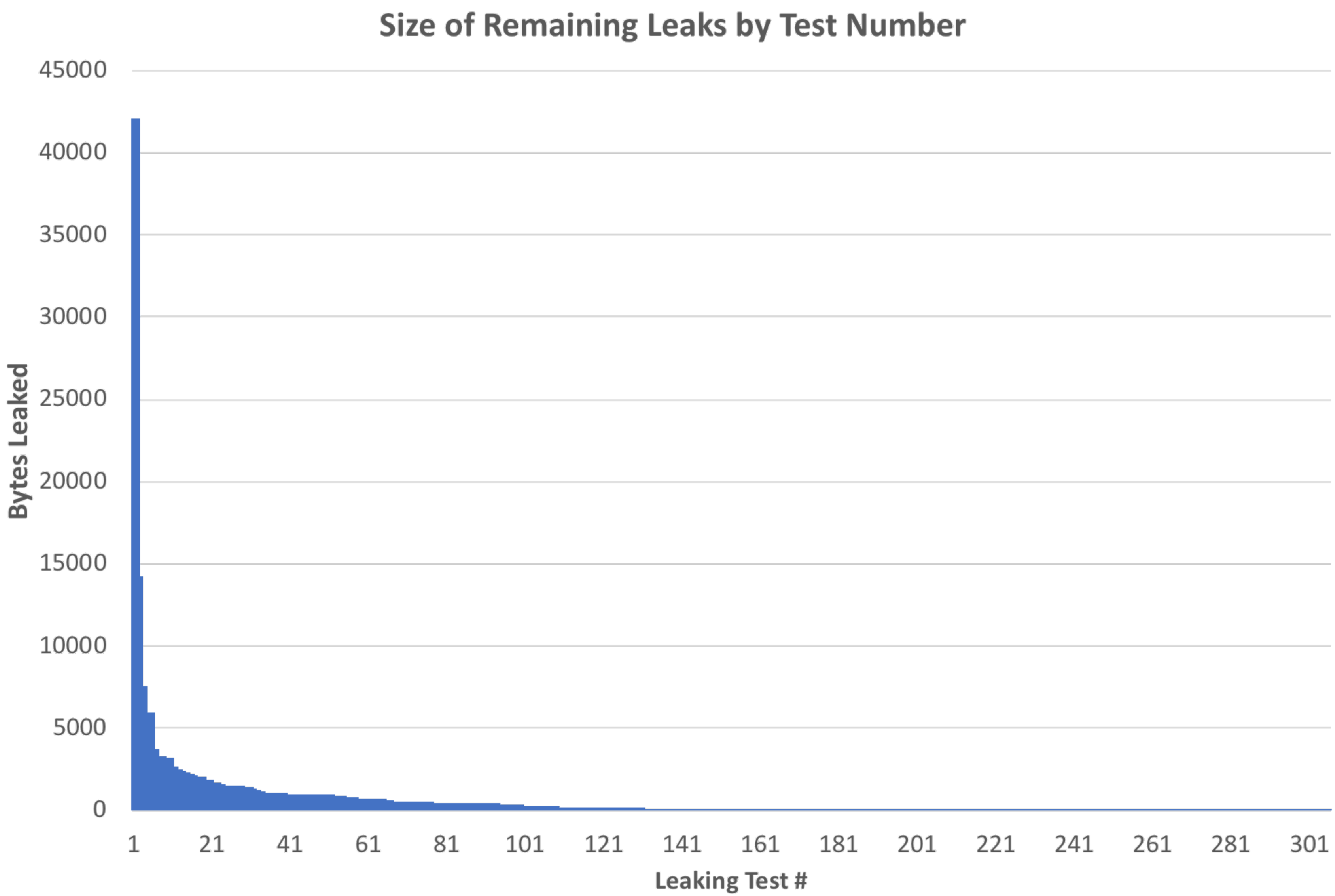


Memory Leaks for All Tests

- **Considering all tests, a similar story but noisier**
  - Spikes typically due to new tests with user-level leaks being added

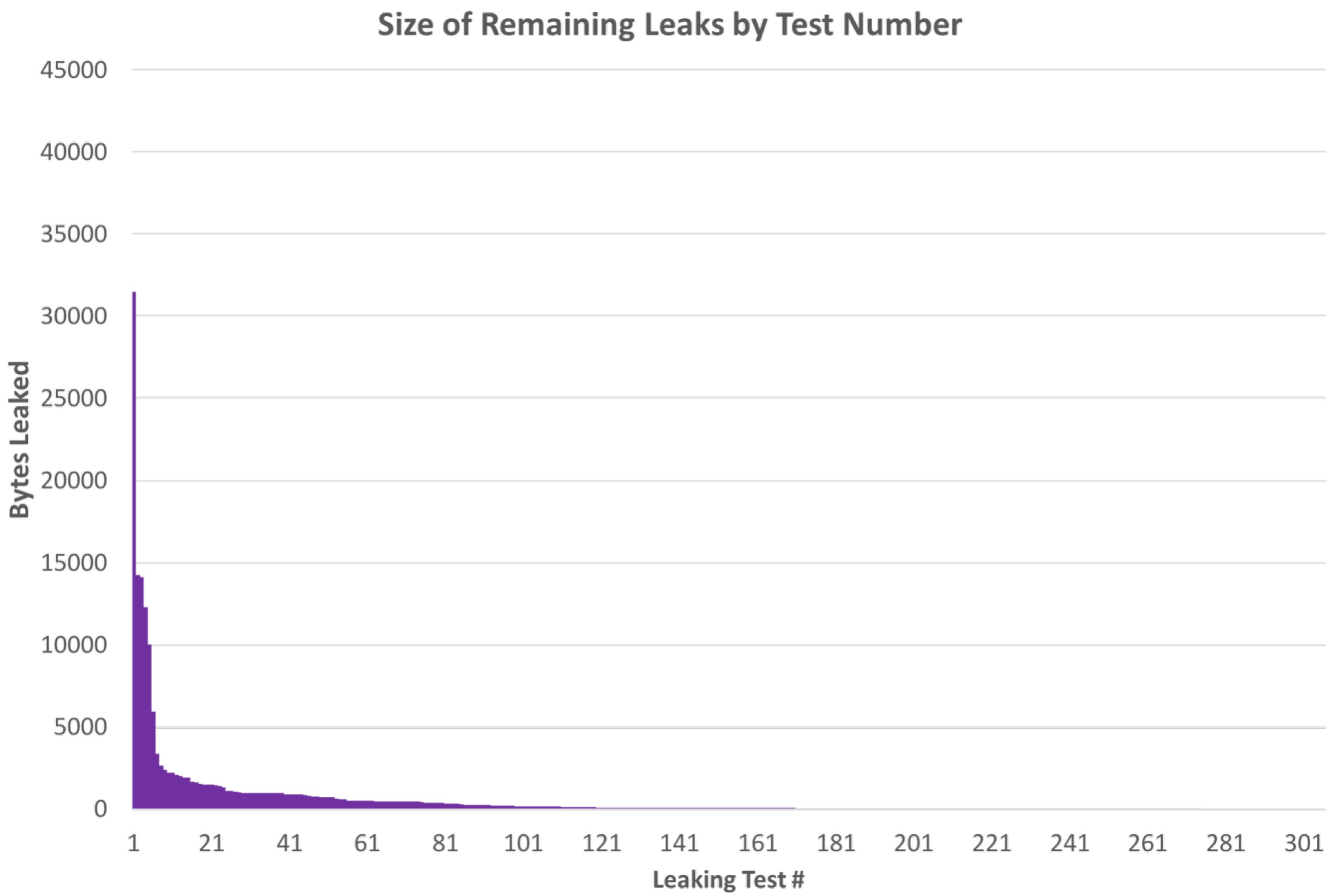


Memory Leaks: Remaining Leaks (as of 1.17)

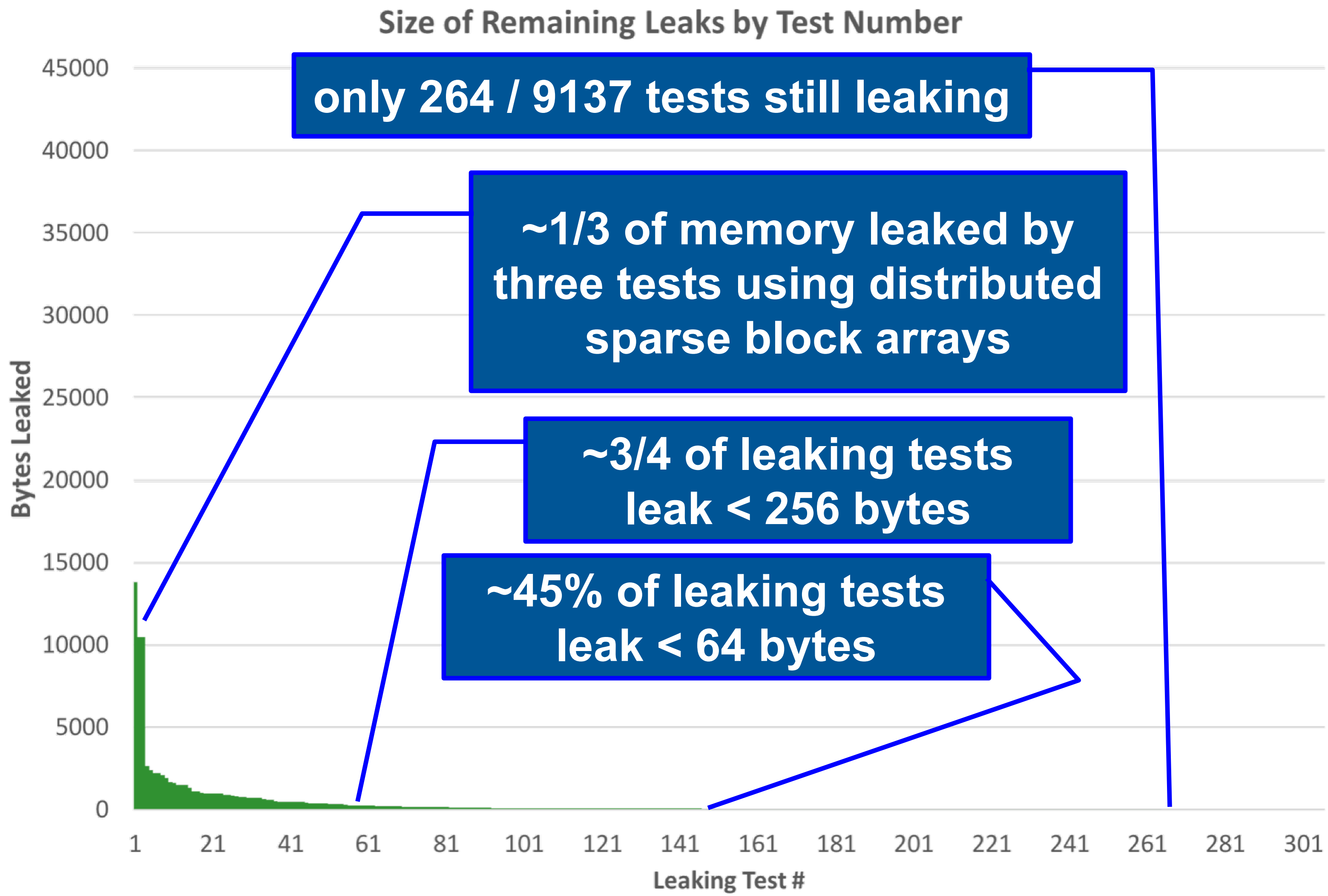


COMPUTE | STORE | ANALYZE

Memory Leaks: Remaining Leaks (as of 1.18)



Memory Leaks: Remaining Leaks (as of Sept 19)



Memory Leaks: Status

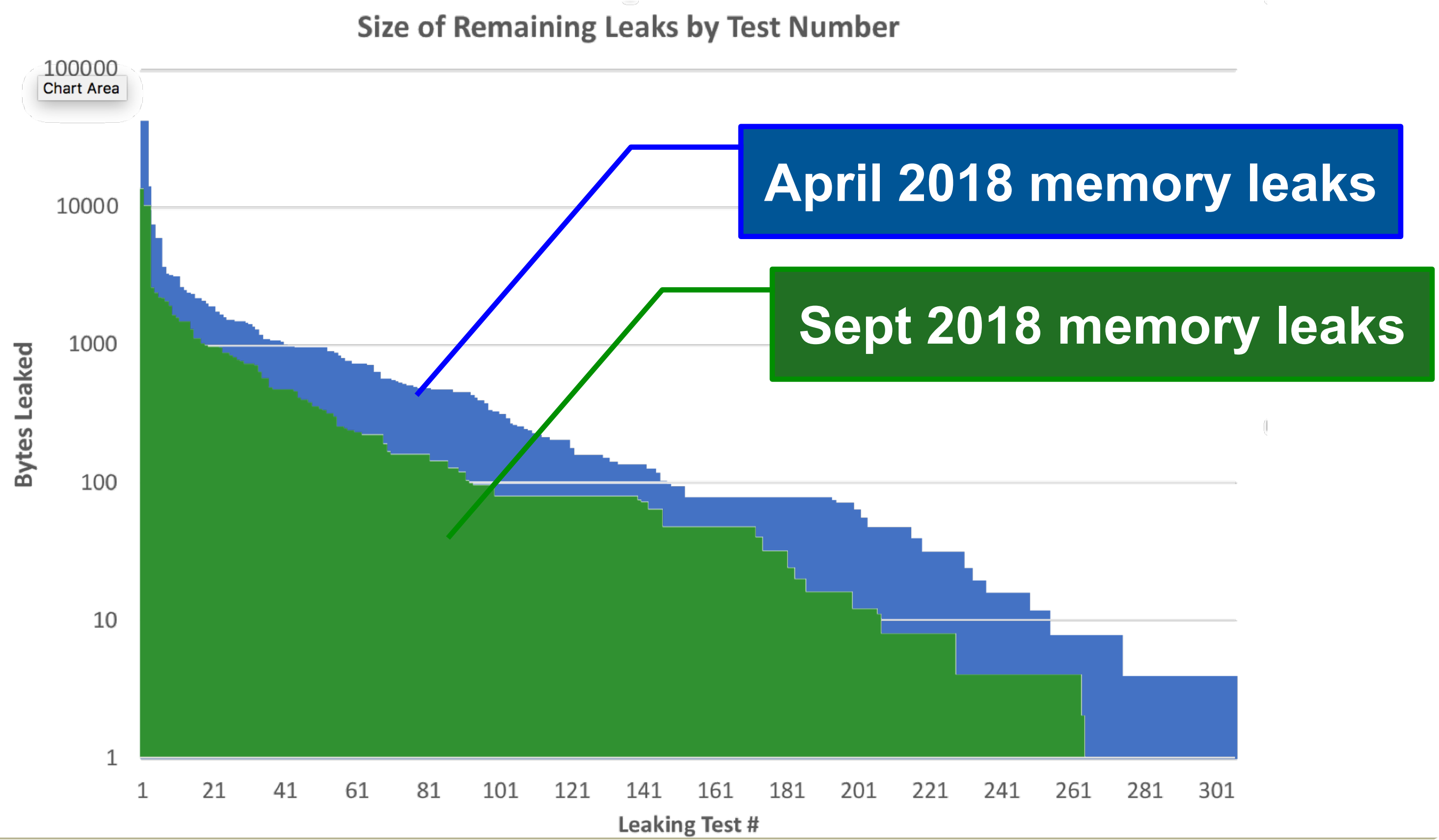
Status:

- From 1.17–1.18, leaks reduced by 25% in testing (w/ ~750 new tests)
  - leaks reduced by 60% compared to 1.17 with sparse domain fix
- Primary known cases of remaining leaks:
  - certain distributed sparse block cases
  - compiler-generated iterator classes in certain cases
  - aspects of global arrays of arrays
  - certain domain map meta-data
  - certain first-class-functions
  - user-level leaks in tests themselves

Memory Leaks: Next Steps

Next Steps:

- Continue working through remaining leaks as a background task
- Once no leaks remain, make addition of new leaks a failure mode



For More Information

For additional optimization and benchmark changes in the 1.18 release, refer to the 'Performance Optimizations', 'Cray-specific Performance Optimizations', 'Memory Improvements', and 'Example Codes' sections in the **CHANGES.md** file.

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