Benchmarks and Performance Optimizations

Chapel version 1.19

March 21, 2019

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

Outline

- Bale Case Study
  - Bale Histogram
  - Bale Indexgather
  - Blocking Comm Optimizations
  - UnorderedCopy
  - Unordered Compiler Optimization
- Parallelizing Scans
- cstdlib Atomics
- Stream Case Study
  - Block Distribution Optimization
  - Remote Task Spawn Optimizations
- Memory Leaks

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)

- Bale also contains aggregated communication libraries
  - Compares elegant/intuitive code vs. more complex aggregated code

Bale: Chapel Background and Effort

Background: Focused on intuitive/elegant implementations in 1.18

- Ported elegant version of histogram and tuned performance
- Added buffered atomics to manually optimize for even better performance
  - At the cost of elegance

This Effort:

- Improved elegance of manually optimized histogram
- Ported elegant version of indexgather and tuned performance
- Added compiler optimization to automatically optimize intuitive versions

Bale Histogram

Histogram: Background

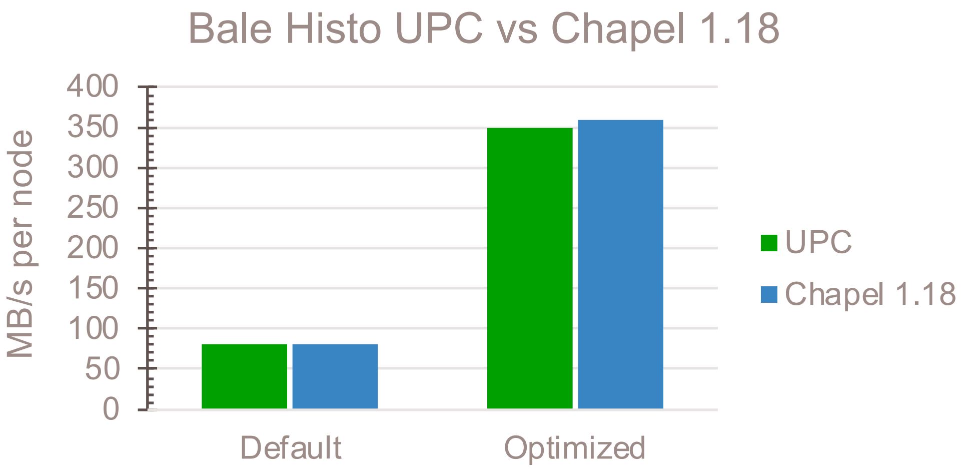
- Optimized version of Histogram in 1.18 was fast, but not very elegant
  - Revealed too much about implementation, required explicit flush

Default Chapel

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

Optimized Chapel

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



Histogram: This Effort

- Renamed BufferedAtomics to UnorderedAtomics
  - For example `addBuff() => unorderedAdd()`
  - Eventually want `add(order=unordered)`
    - Not quite ready to commit to language-level API, but one step closer
- Added implicit fences at task/forall termination, explicit fence no longer needed

Optimized Chapel 1.18

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

Optimized Chapel 1.19

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

Histogram: Impact

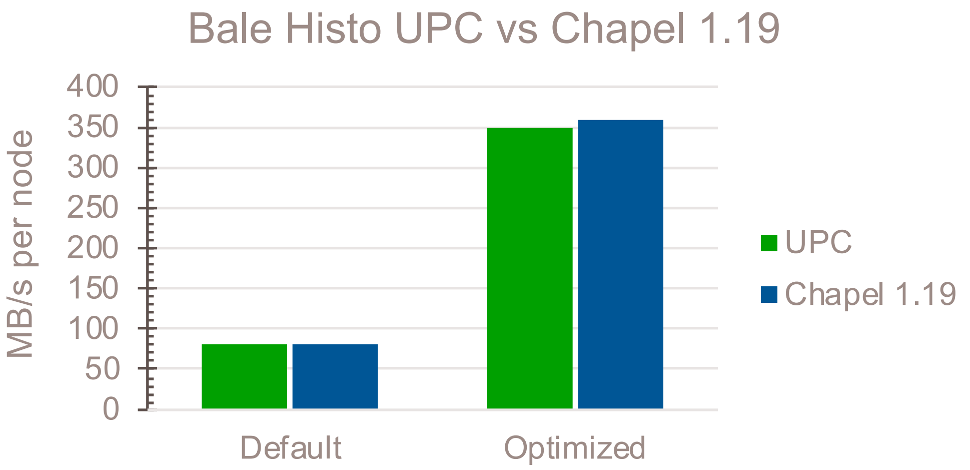
- Optimized histogram implementation is more elegant
  - Performance still on par with reference UPC

Default Chapel

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

Optimized Chapel

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



Bale Indexgather

Indexgather: Background

- Indexgather does random GETs from a distributed array

Default UPC

```
for(i = 0; i < T; i++) {
  tgt[i] = lgp_get_int64(table, index[i]);
}
```

Default Chapel

```
forall i in rindex.domain {
  tgt[i] = A[rindex[i]];
}
```

Optimized UPC

```
for(i = 0; i < T; i++){
  #pragma pgas defer_sync
  tgt[i] = lgp_get_int64(table, index[i]);
}
lgp_barrier();
```

Indexgather: Background

- By default, remote operations in Chapel are "blocking"
  - Supports Memory Consistency Model (MCM)
    - "sequential consistency for data-race-free programs"

```
var a = 1;
on Locales[1] {
  var b = a;
  writeln(b); // must print 1
}
```

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

Indexgather: Background

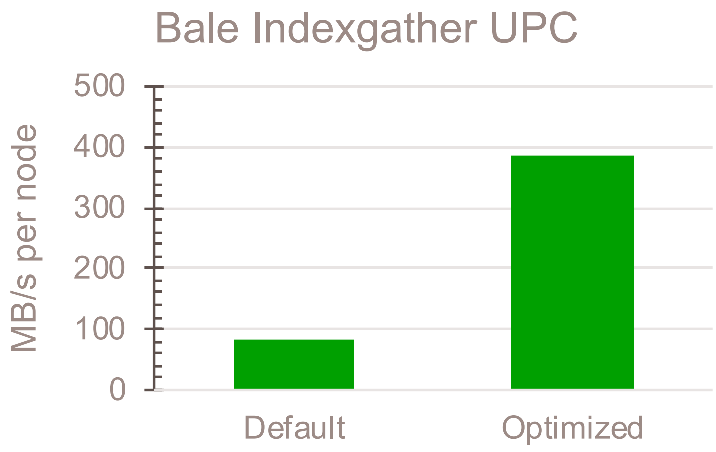
- Cray UPC/SHMEM can drop to more relaxed MCM modes
  - “Use the ‘pgas defer_sync’ directive to force the next statement to be non-blocking”

Default UPC

```
for(i = 0; i < T; i++) {
  tgt[i] = lgp_get_int64(table, index[i]);
}
```

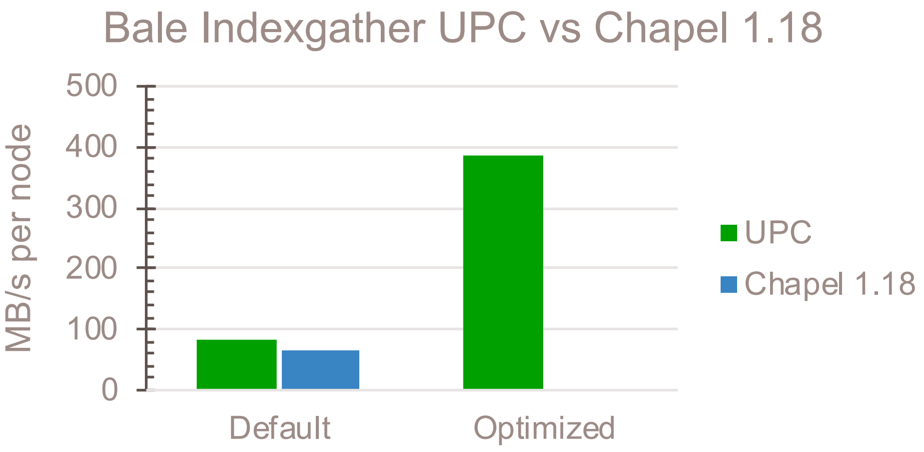
Optimized UPC

```
for(i = 0; i < T; i++){
  #pragma pgas defer_sync
  tgt[i] = lgp_get_int64(table, index[i]);
}
lgp_barrier();
```



Indexgather: Background

- Chapel performance lagged behind reference UPC
  - ~20% off for default case, 5x off for optimized



Blocking Comm Optimization

Blocking Comm: Background

- In 1.18 ugni yielded continuously while waiting for ACK for GETs/PUTs
  - Yielding allows for comm/compute overlap
  - Yield slower 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
```

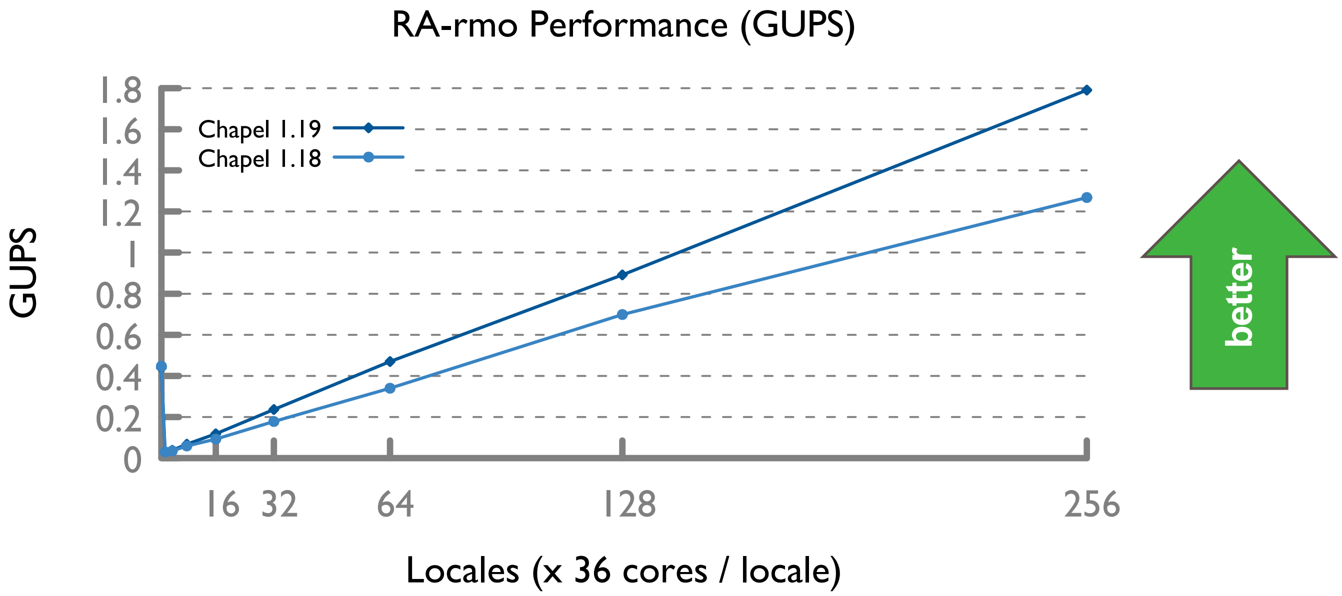
Blocking Comm: This Effort

- Optimized blocking GETs/PUTs by only 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, 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
  consume_all_outstanding_cq_events(cdi);
} while (!atomic_load_bool(&post_done));         // blocking wait for transaction to complete
```

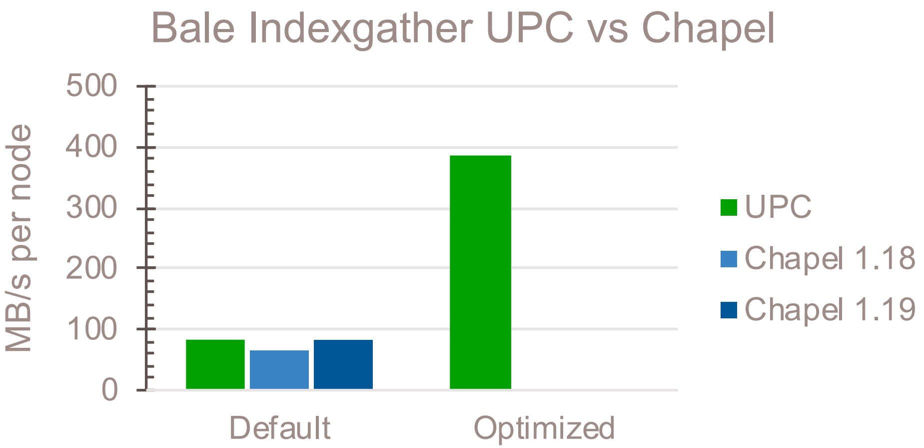
Blocking Comm: Performance Impact

- Improved performance of blocking GETs/PUTs



Blocking Comm: Indexgather Impact

- Default Indexgather performance on par with UPC



UnorderedCopy

UnorderedCopy: Background

- Remote copies in Chapel were always ordered/blocking
  - Straightforward way to implement sequential consistency

```
var a = 1;
on Locales[1] {
  var b = a;
  writeln(b);  // must print 1
}
```

- Blocking operations limit network injection rate
  - This is why Chapel was slower than the “defer_sync” UPC version

UnorderedCopy: This Effort

- Added UnorderedCopy to permit more relaxed copies
  - Operations are not consistent with normal operations
  - Implicitly fenced at task/forall termination, and can be explicitly fenced

```
var a = 1;
on Locales[1] {
  var b: int;
  unorderedCopy(b, a);
  writeln(b);              // can print 0 or 1
  unorderedCopyFence();
  writeln(b);              // must print 1
}
```

UnorderedCopy: Impact

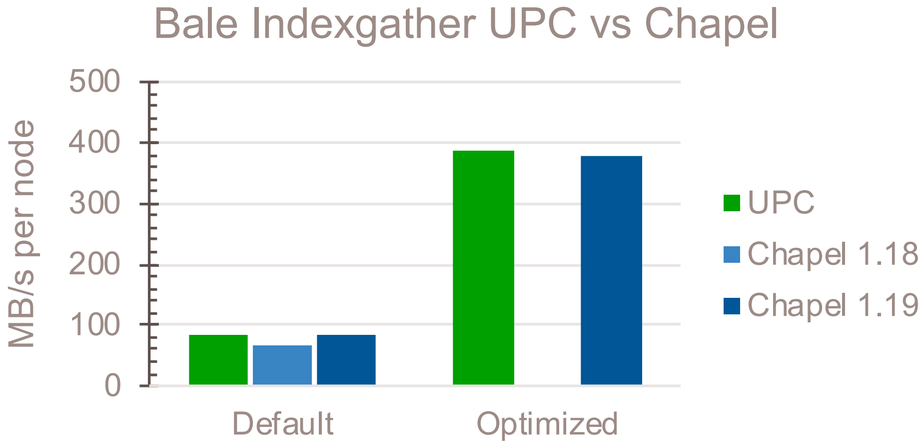
- Permits optimized Indexgather that performs on par with optimized UPC

Default Chapel

```
forall i in rindex.domain {
  tgt[i] = A[rindex[i]];
}
```

Optimized Chapel

```
forall i in rindex.domain {
  unorderedCopy(tgt[i], A[rindex[i]]);
}
```



Unordered Compiler Optimization

Unordered Compiler Opt: Background

- Previous slides showed manual optimizations for Histogram and Indexgather
  - But the compiler should be able to automatically optimize when …
    - Inside a forall loop (no ordering requirements across iterations)
    - Lifetime of operands is longer than forall loop scope
    - Operations are not used for synchronization
    - Result of operation is not used within the same iteration

Histogram

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

Indexgather

```
forall i in rindex.domain do
  tgt[i] = A[rindex[i]];
```

Unordered Compiler Opt: This Effort

- Added compiler optimization to automatically use unordered operations
  - Off by default, can be enabled with `--optimize-forall-unordered-ops`
  - Legal for synchronization-free loops that don't maintain state across iterations

- Implementation strategy:
  - Uses lifetime analysis to ensure operands outlive forall scope
  - Skip loops with optimization hazards (atomic loops, task private vars, etc.)
  - Transform operations in last statement to unordered operations
    - Last statement rule ensures result it not used later in the same iteration

Unordered Compiler Opt: Impact

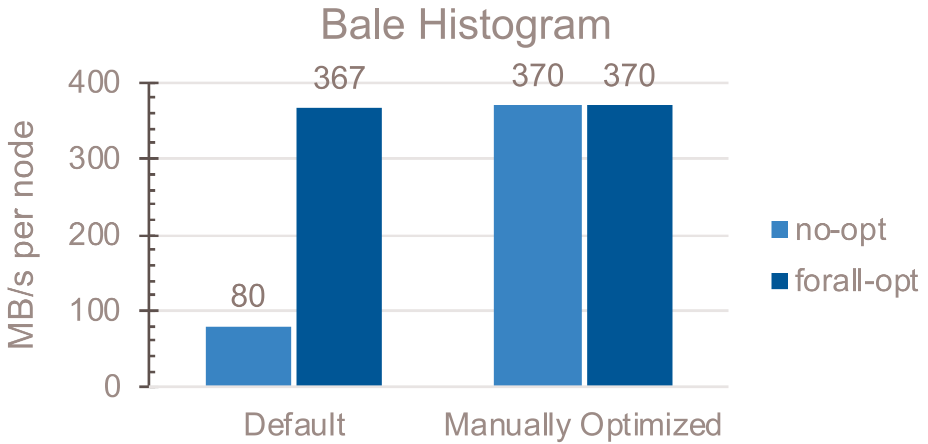
- Default Histogram nearly on par with performance of manually optimized version
  - Compiler adds local memory fence to maintain consistency of local operations

Default Histogram

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

Manually Optimized Histogram

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



Unordered Compiler Opt: Impact

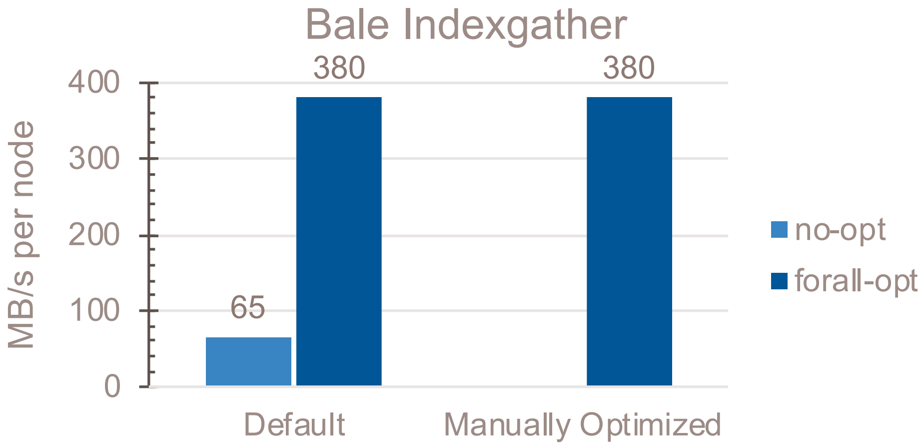
- Default Indexgather on par with manually optimized version

Default Indexgather

```
forall i in rindex.domain do
  tgt[i] = A[rindex[i]];
```

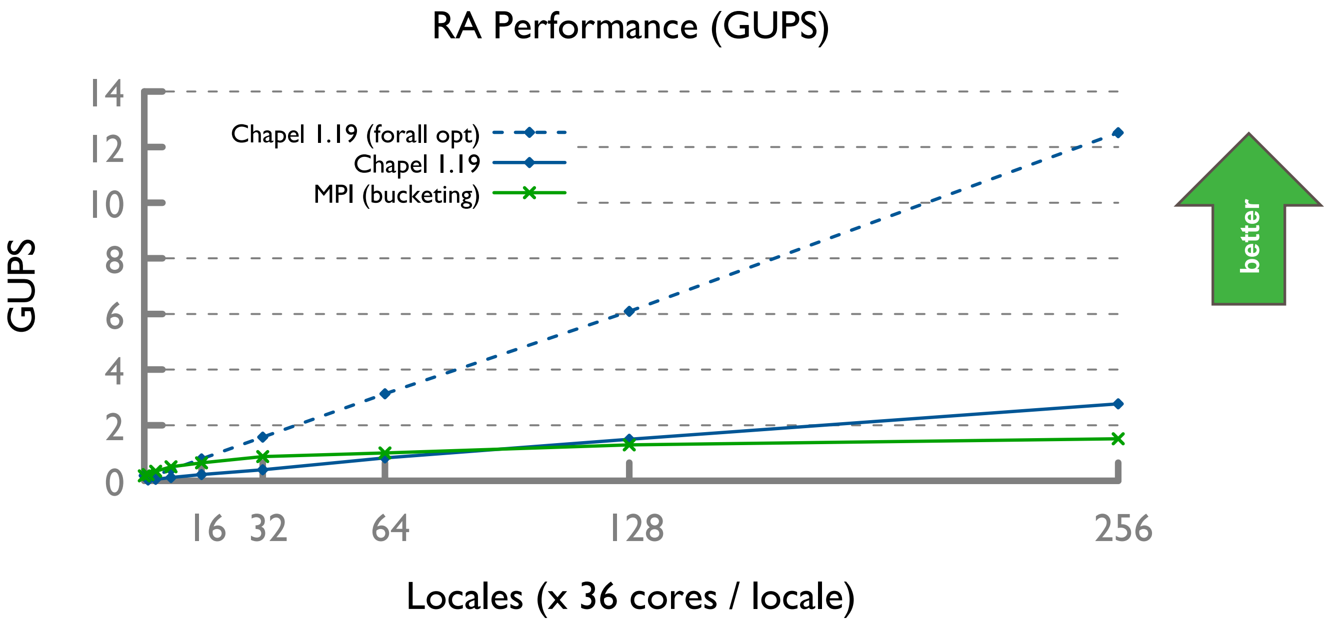
Manually Optimized Indexgather

```
forall i in rindex.domain do
  unorderedCopy(tgt[i], A[rindex[i]]);
```



Unordered Compiler Opt: Impact on HPCC RA

- Optimization also improves atomic RA version by 4x *with no source changes*



Unordered Compiler Opt: Next Steps

- Enable compiler optimization by default
- Consider extending optimization beyond last statement in a forall
  - Will require additional analysis (dataflow, alias analysis)

```
forall r in rindex {
  A[r] = B[r];
  C[r] = D[r];
}
```

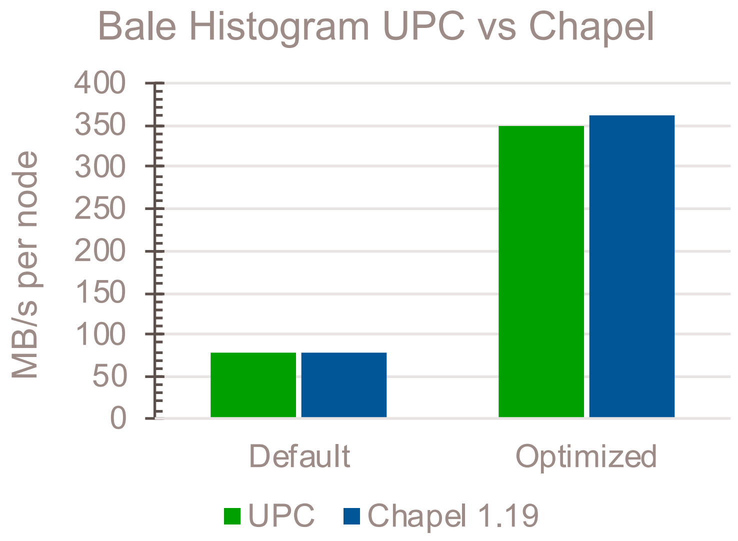
Bale Summary

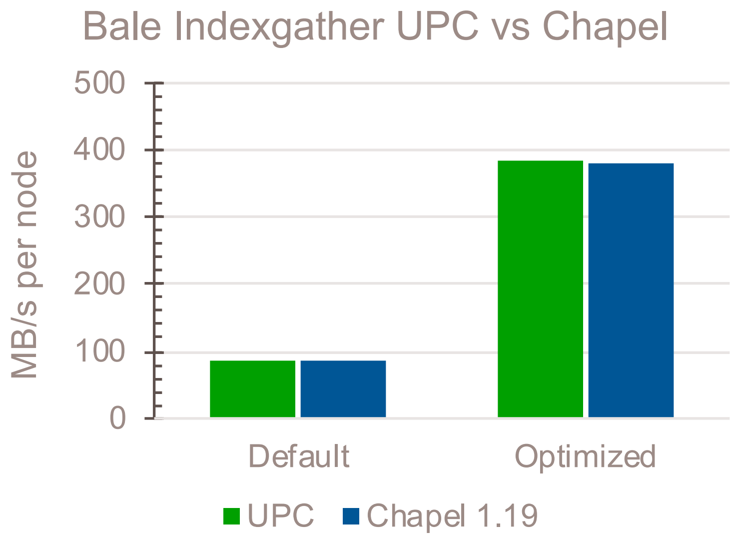
Bale: Summary

- Improved elegance of manually optimized Bale Histogram
  - Renamed BufferedAtomics to UnorderedAtomics
  - Unordered operations are flushed at task/forall termination
- Ported and improved performance of Bale Indexgather
  - Improved performance of blocking GETs/PUTs
  - Added UnorderedCopy to further optimize GETs/PUTs
- Added prototype compiler optimization to automatically use unordered operations
  - Allows default versions to achieve same performance as manually optimized

Bale: Performance Summary

- Performance of Bale Histogram and Indexgather on par with UPC





Bale: Next Steps

- Flesh out remaining unordered operation API
  - Fetching Atomics (including Compare-and-Swap)
- Improve unordered compiler optimization
  - Enable it by default
  - Relax last statement restriction
- Port and tune more Bale Applications
- Start writing aggregated versions of benchmarks
  - Current effort focused on making non-aggregated as fast as possible
  - But still large speedup possible from doing aggregation

Parallelizing Scans

Parallel Scans: Background

- Chapel has long supported a scan operator
  - sibling to `reduce`, designed to support parallel prefix operations

```
var offset = + scan bytesPerElem;
```

- However, its implementation has always been serial and locality-oblivious

```
$ chpl myProg.chpl
myProg.chpl:2: warning: scan has been serialized (see issue #5760)
$ ./myProg   # slooooow!
```

Parallel Scans: This Effort

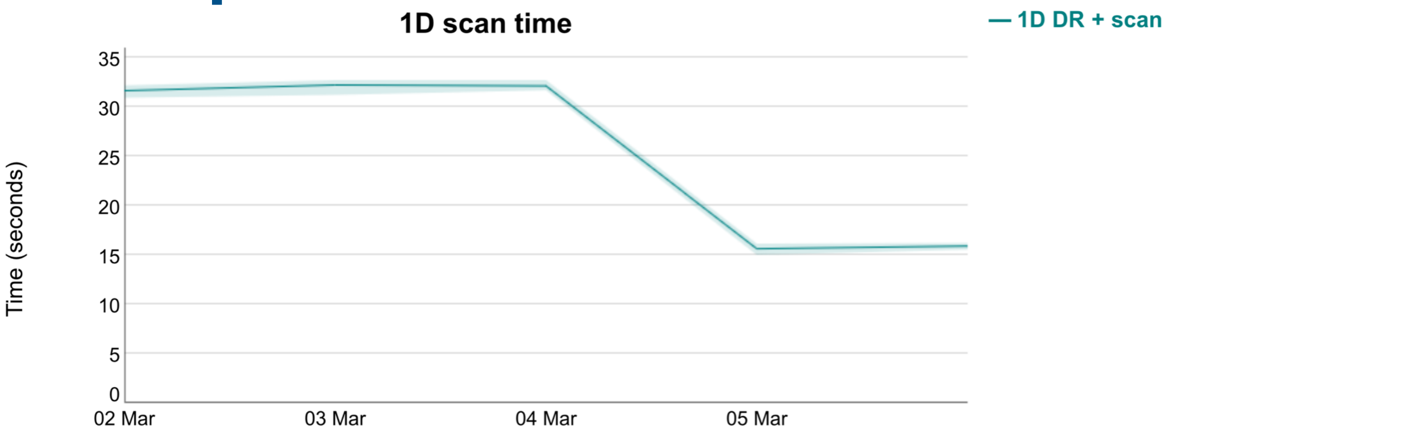
- Parallelized scans of local and block-distributed arrays
  - implemented as part of a domain map's optional interface
- Made this an opt-in feature due to its late addition to the release
- Warning updated to flag cases when it could be applied

```
$ chpl myProg.chpl
myProg.chpl:2: warning: scan has been serialized (see issue #5760)
myProg.chpl:2: warning: (recompile with -senableParScan to enable
a prototype parallel implementation)
$ chpl myProg.chpl -senableParScan
$ ./myProg   # fast!
```

Parallel Scans: Impact

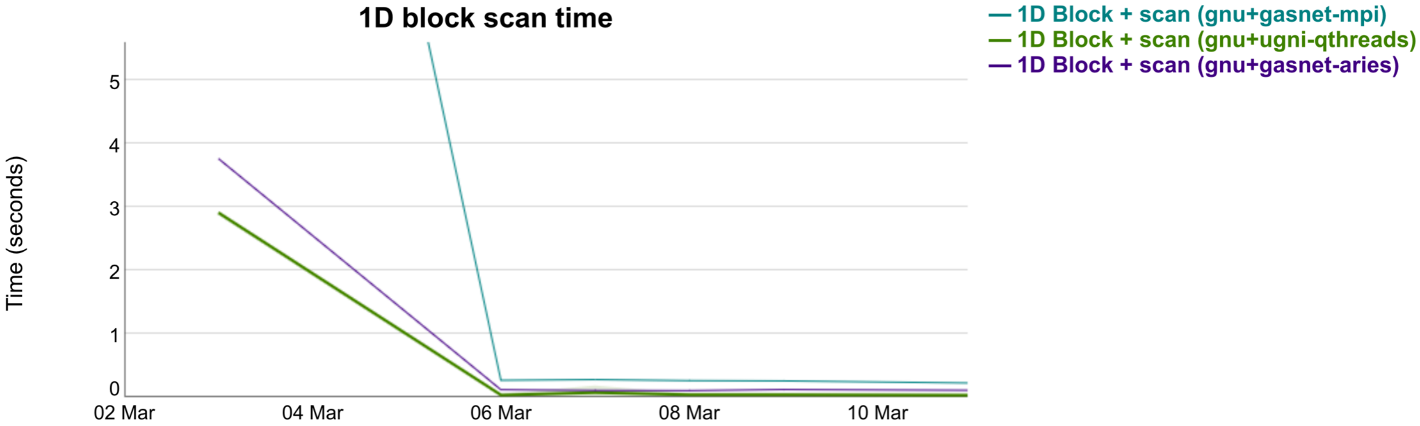
Local array:

- 24 cores
- ¼ memory (~8B elements)



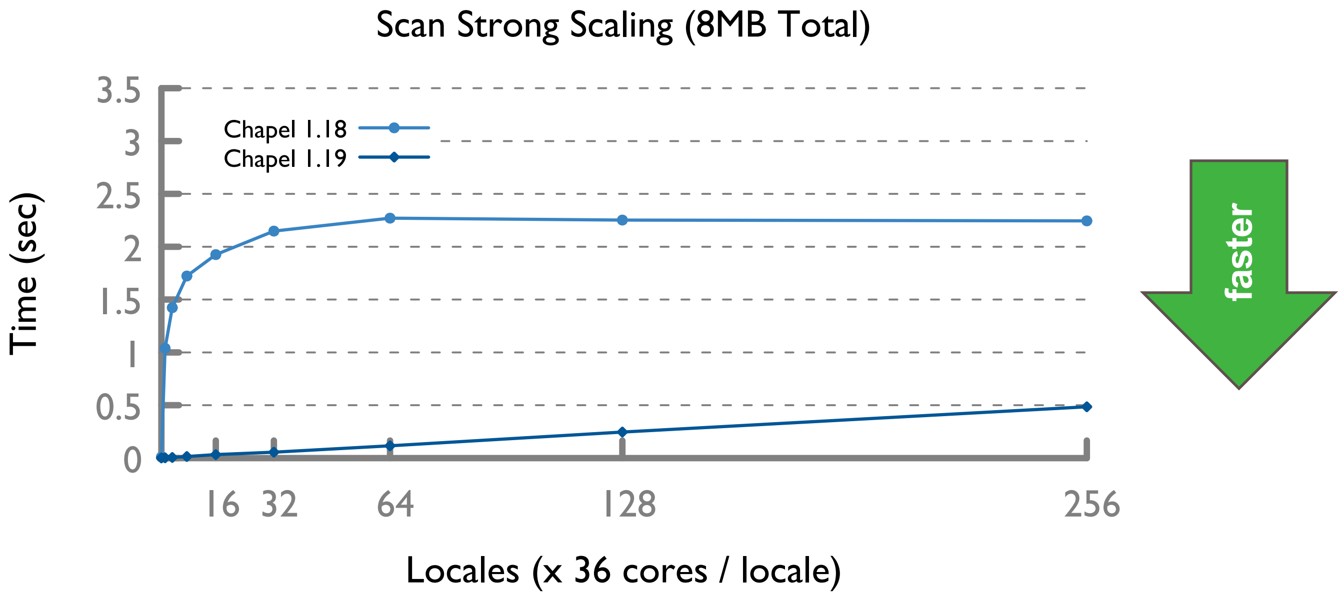
Block array:

- 16 locales x 28 cores
- 1M elements



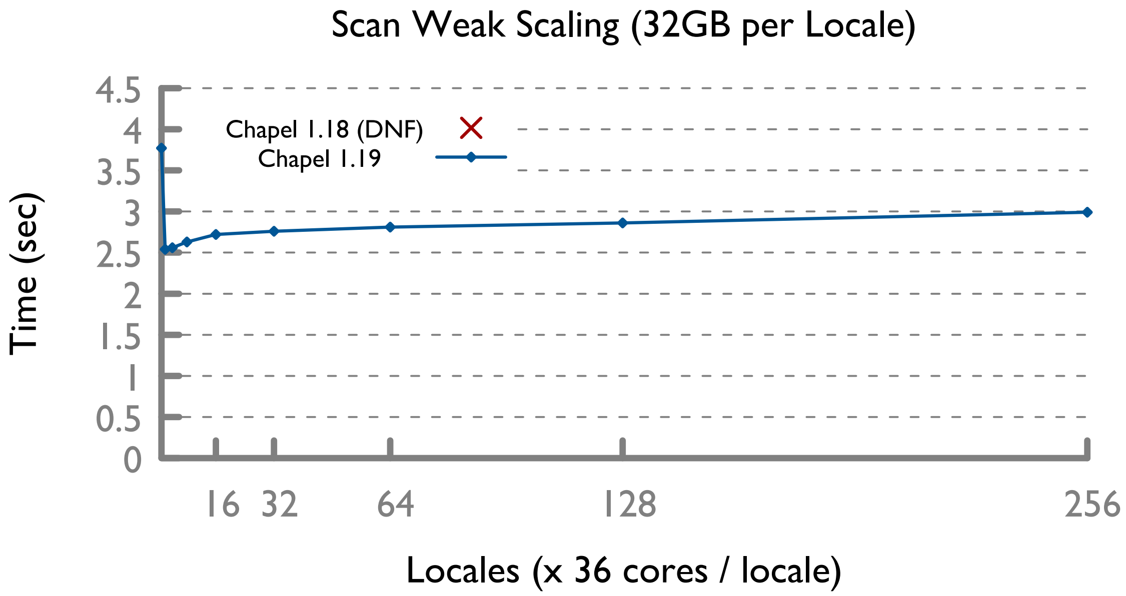
Parallel Scans: Impact (Strong Scaling)

- Significantly improved scalability



Parallel Scans: Impact (Weak Scaling)

- Significantly improved scalability



Parallel Scans: Next Steps

- Enable the parallel implementation by default
- Extend support to array expressions that preserve shape / domain
  - e.g., `const perm = + scan mask: int;`
- Implement for additional distributions
- Consider making the default implementation parallel, distributed (locality-oblivious?)

cstdlib Atomics

cstdlib Atomics: Background

- Chapel has 3 atomic implementations:
  - locks -- uses pthread mutexes to implement atomics
    - portable but very slow, only default under PGI
  - intrinsics -- uses __sync compiler intrinsics to implement atomics
    - mostly fast, but memory orders ignored and read() is slow
      (no read intrinsic, implemented with CAS for portability)
    - fairly portable, previously default everywhere except PGI
  - cstdlib -- uses C11 atomics to implement atomics
    - best performance, memory orders adhered to
    - becoming more portable (GCC 5, modern Clang, Intel 18, Cray 8.7.7)

cstdlib Atomics: Background

- Want cstdlib atomics to be the default
  - Designed to map efficiently to a wide range of hardware
  - Supports relaxed memory orders
  - Has well-defined semantics

- Tried to make cstdlib atomics the default in 1.14, but ran into several problems
  - Performance regressions for GCC
  - Portability issues for Clang and LLVM
  - Lack of support from Intel and Cray compilers

cstdlib Atomics: This Effort

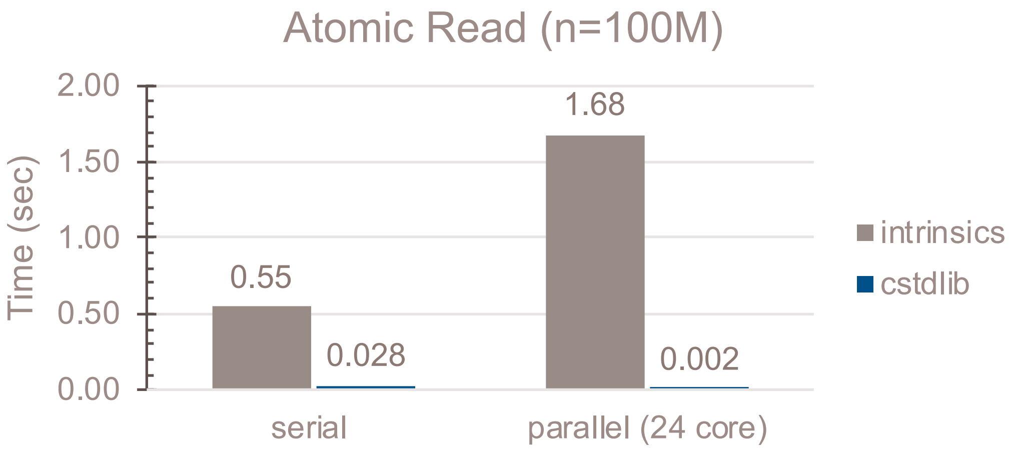
- Made cstdlib atomics default for GCC >= 5, Clang with feature detection, LLVM
  - Made test and set locks use acquire/release to avoid GCC performance hit
  - Worked around Clang portability issue when system header is from GCC < 5
  - Fixed LLVM support by avoiding macro definitions of atomic_thread_fence()

- Verified cstdlib atomics work with Intel 18, Cray 8.7.7
  - Not quite ready to make it the default for these compilers
    - Chapel module on Crays built with older versions (for ABI compatibility)

cstdlib Atomics: Performance Impact

- Significantly faster reads, concurrency now results in speedup not slowdown
  - 20x serial speedup, 800x parallel speedup

```
for 1..n do                    forall 1..n do
  a.read();                      a.read();
```

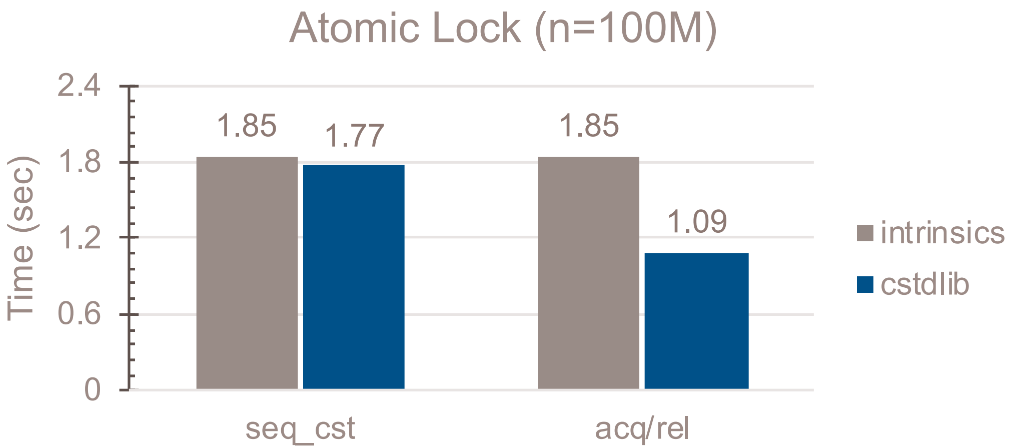


cstdlib Atomics: Performance Impact

- Memory order optimizations possible

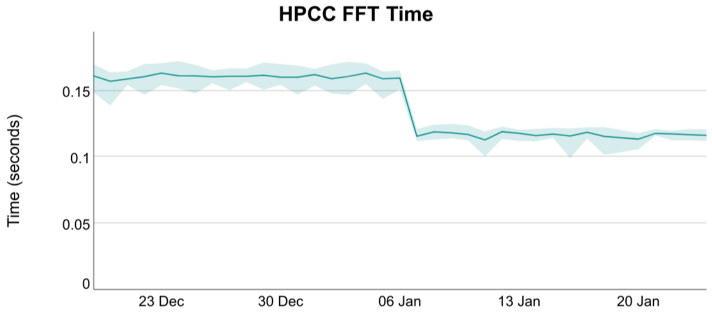
```
for 1..n {
  while l.testAndSet() {}
  l.clear();
}
```

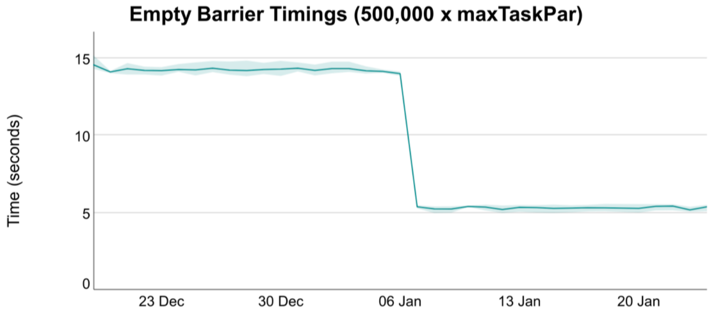
```
for 1..n {
  while l.testAndSet(acquire) {}
  l.clear(release);
}
```

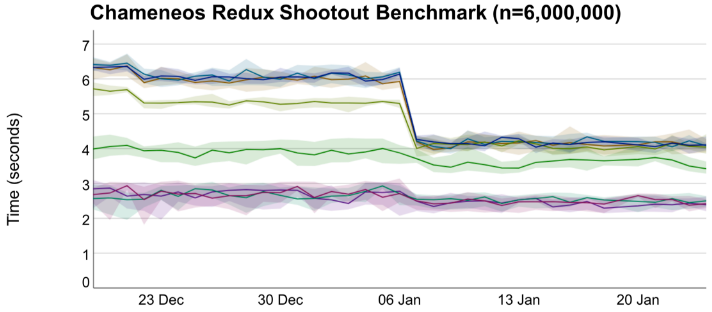


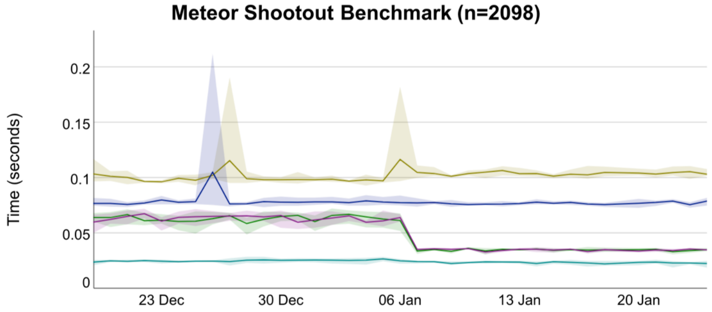
cstdlib Atomics: Performance Impact

- Benchmarks using relaxed atomics improved









cstdlib Atomics: Next Steps

- Make cstdlib atomics the default for Intel and Cray compilers

- Use more relaxed memory orderings for core idioms when possible
  - counters
  - internal locks
  - reference counting

Stream Case Study

Stream: Background

- Multiple variants of Stream benchmark exist, e.g.:
  - **EP:** Explicit SPMD, uses local arrays, task spawning not included in time
  - **Global:** Elegant, uses block distributed arrays, task spawning included in time

Stream EP

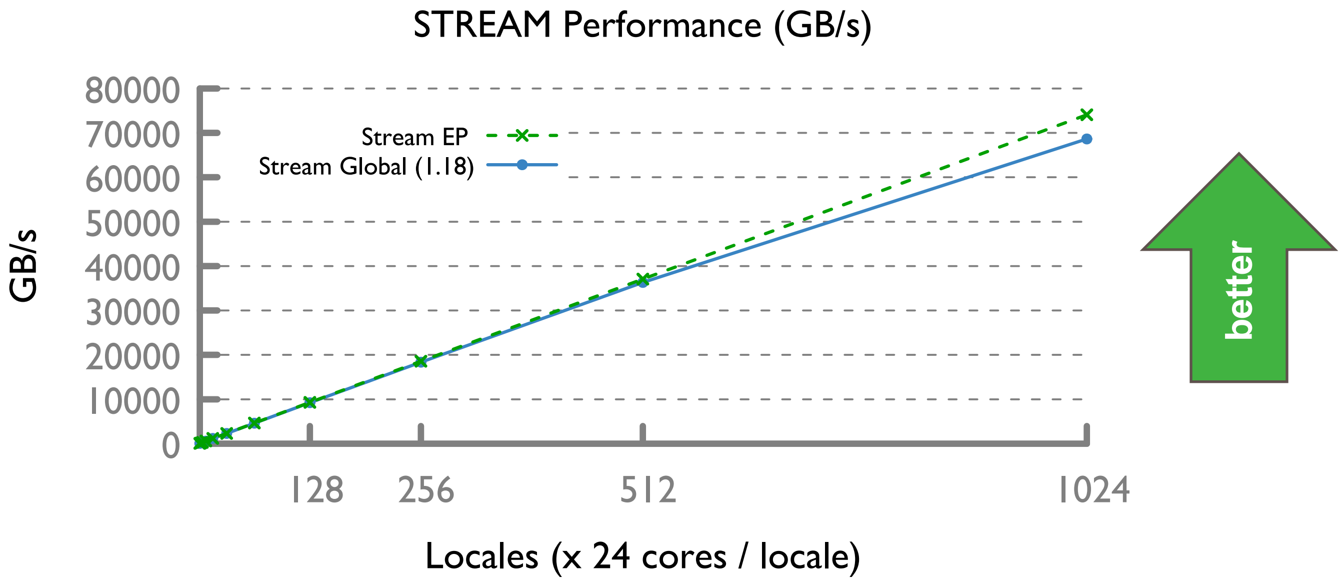
```
coforall loc in Locales do on loc {
  var A, B, C: [1..m] real;
  initVectors(B, C);

  startTimer();

  forall (a, b, c) in zip(A, B, C) do
    a = b + alpha * c;

  stopTimer();
}
```

Global Stream

```
const Space = {1..m} dmapped Block({1..m});
var A, B, C: [Space] real;
initVectors(B, C);

startTimer();

forall (a, b, c) in zip(A, B, C) do
  a = b + alpha * c;

stopTimer();
```

Stream: Background

- In 1.18, Global Stream performance lagged at higher locale counts



Stream: This Effort

- For 1.19, Global Stream performance and scalability were improved by
  - Optimizing iteration over block-distributed arrays
  - Improving remote task-spawning speed

Block Distribution Optimization

Block Distribution: Background

- Iteration over block distribution was suboptimal, compiler could not prove locality

```
forall (a, b, c) in zip(A, B, C) do
  a = b + alpha * c;
```

Generated:

```
// compute &C[i]
wide_c_ptr_i.locale = chpl_gen_getLocaleID();
wide_c_ptr_i.addr = (wide_c_ptr->addr + i3);
// compute &B[i]
wide_b_ptr_i.locale = chpl_gen_getLocaleID();
wide_b_ptr_i.addr = (wide_b_ptr->addr + i2);
// compute &A[i]
wide_a_ptr_i.locale = chpl_gen_getLocaleID();
wide_a_ptr_i.addr = (wide_a_ptr->addr + i1);
```

```
// "local" gets (short-circuited in runtime)
chpl_gen_comm_get(local_c,
wide_c_ptr_i.locale, wide_c_ptr_i.addr);
chpl_gen_comm_get(local_b,
wide_b_ptr_i.locale, wide_b_ptr_i.addr);
// computation
tmp_comp = local_b + alpha * local_c;
// "local" put (short-circuited in runtime)
chpl_gen_comm_put(tmp_comp,
wide_a_ptr_i.locale,  wide_a_ptr_i.addr);
```

Block Distribution: Effort and Impact

- Manually narrowed local array in iterator, significantly improved generated code

```
forall (a, b, c) in zip(A, B, C) do
  a = b + alpha * c;
```

Now generates:

```
c_ptr_i = (c_ptr + i3);
b_ptr_i = (b_ptr + i2);
a_ptr_i = (a_ptr + i1);
*(a_ptr_i) = (*b_ptr_i) + alpha * (*c_ptr_i);
```

- 2% performance improvement for Stream Global at 256 locales

ugni: Remote Task Spawn Optimizations

Blocking Task Spawn: Background

- Task creation and on-statements are used to create remote tasks
- A common idiom is to create a task on each locale

```
coforall loc in Locales do on loc { body(args); }
```

Blocking Task Spawn: Background

- Under ‘ugni’ in 1.18, remote-coforalls were translated into something like:

```
var endCount: atomic int = Locales.size;
for loc in Locales {
  var ACK = startRemoteTask(loc, bodyWrap, args, endCount,);
  while (!received(ACK)) {}



}
endCount.waitFor(0);

proc bodyWrap(args, endCount) { body(args); endCount.sub(1); }
```

Blocking Task Spawn: Background

- Under ‘ugni’ in 1.18, remote-coforalls were translated into something like:

```
var endCount: atomic int = Locales.size;
for loc in Locales {
  var ACK = startRemoteTask(loc, bodyWrap, args, endCount,);
  while (!received(ACK)) {} // problem, network round trip wait



}
endCount.waitFor(0);


proc bodyWrap(args, endCount) { body(args); endCount.sub(1); }
```

Blocking Task Spawn: This Effort

- They are now translated into something like:

```
var endCount: atomic int = Locales.size;
for loc in Locales {
  var ACK = startRemoteTask(loc, bodyWrap, args, endCount,);
  ackBuff[ackIndex()] = ACK;
  if ackBuff.full() then          // normally not full, so no waiting
    retireAtLeastOneTX();         // fast, usually a few ready to retire
}
endCount.waitFor(0);

proc bodyWrap(args, endCount) { body(args); endCount.sub(1); }
```

Argument Size: Background

- ‘ugni’ starts remote tasks by issuing a network PUT containing task metadata

- Task arguments (“arg bundles”) need to be available on the remote node
  - Small arg bundles can be sent with initiating PUT, no extra comm required
  - Large arg bundles require additional work and comm
    - Initiating node copies, remote node GETs and then spawns task to free

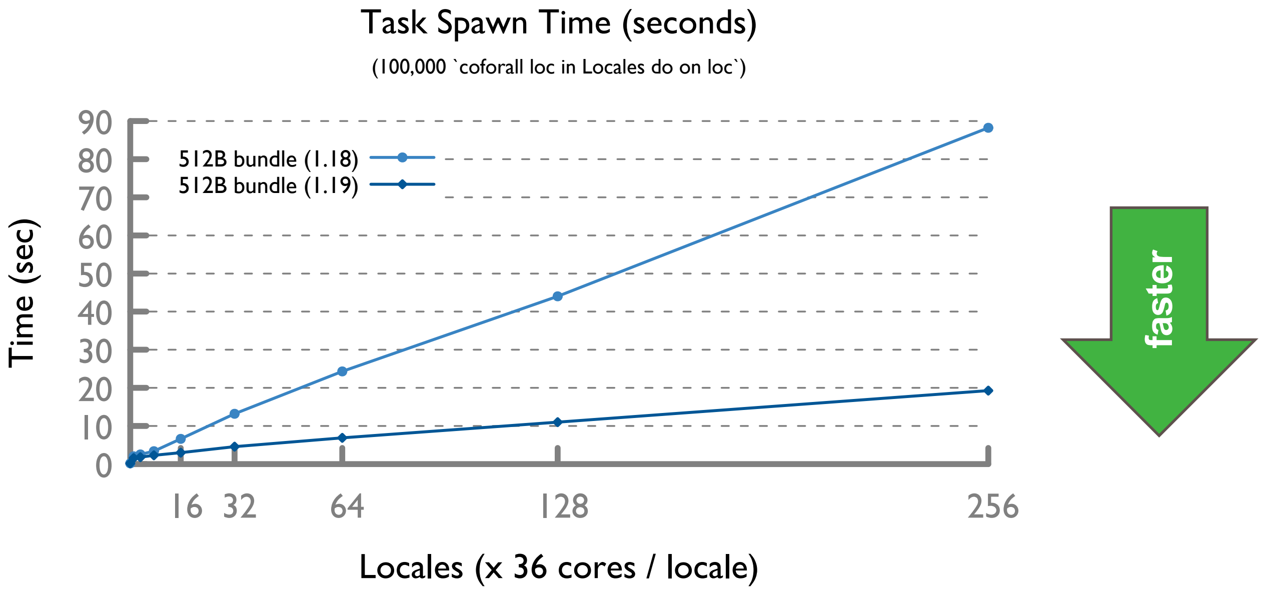
Argument Size: Effort and Impact

- Increased threshold for small arg bundles
  - Increased from 64 Bytes to 1,024 Bytes
  - Typical payload is ~512 Bytes, largest in all benchmarks is 784 Bytes
    - Most remote tasks can now be initiated with a single PUT

  - Memory footprint for task spawning space increased:
    - 128 MB per locale at 1,024 locales (acceptable)
    - 2048 MB per locale at 8,096 locales (high, but no users at this scale yet)

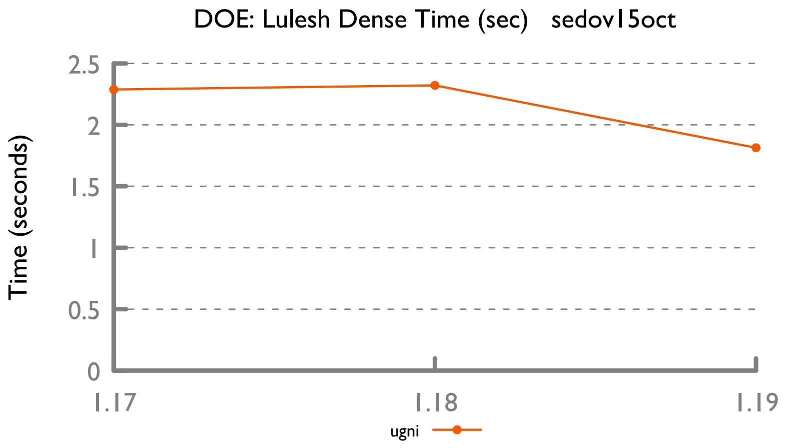
Remote Task Spawn: Performance Impact

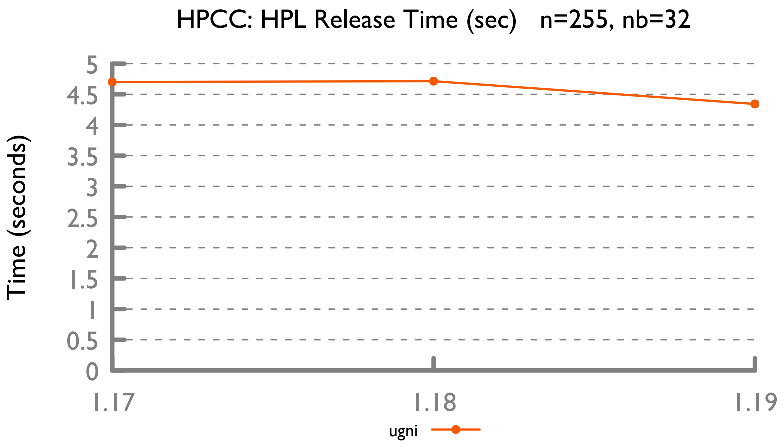
- Remote task spawning is significantly faster with these optimizations
  - 4.5x speedup for a typical task



Remote Task Spawn: Performance Impact

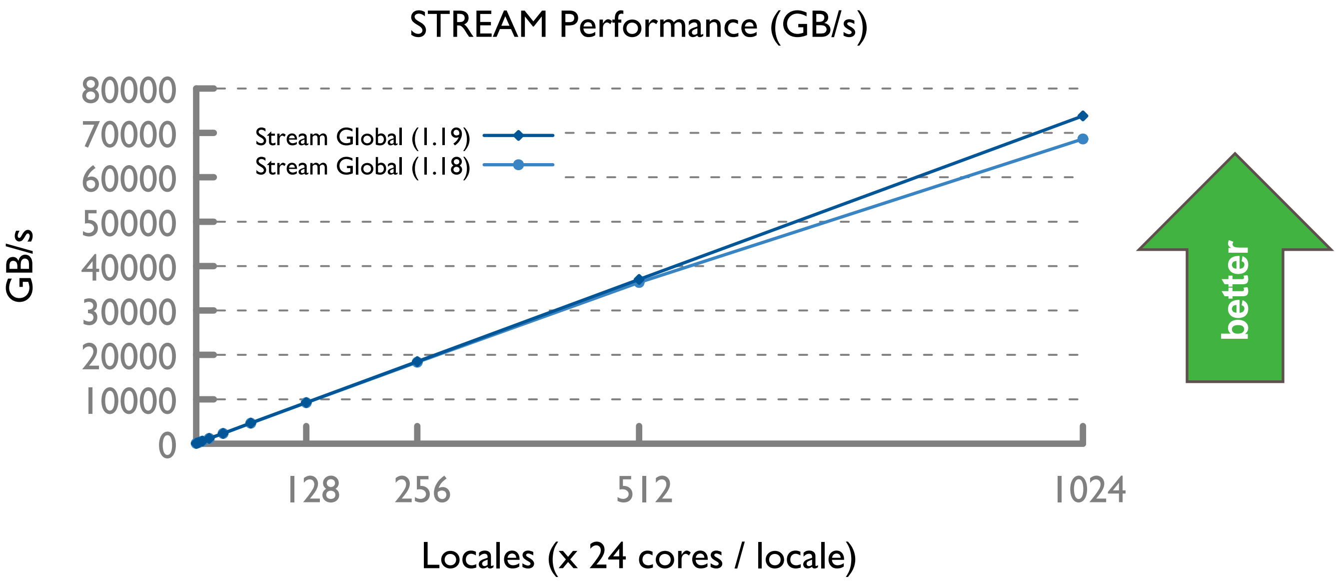
- Speedups for task-heavy benchmarks at small scale (16 locales)





Remote Task Spawn: Performance Impact

- Scalability improvements for benchmarks at larger scales (1,024 locales)



Remote Task Spawn: Next Steps

- One-to-many spawning is nearly as fast as possible for small argument sizes
  - 85% of time spent in uGNI call
  - Rest spent serializing/deserializing arguments and manipulating end count

- Future optimizations will require hierarchical spawning (e.g. tree-based spawns)
  - However, current scheme should be fast enough for at least 4,096 locales

- At higher scales will also have to tune memory footprint
  - Likely optimizing how tasks with large payloads are created

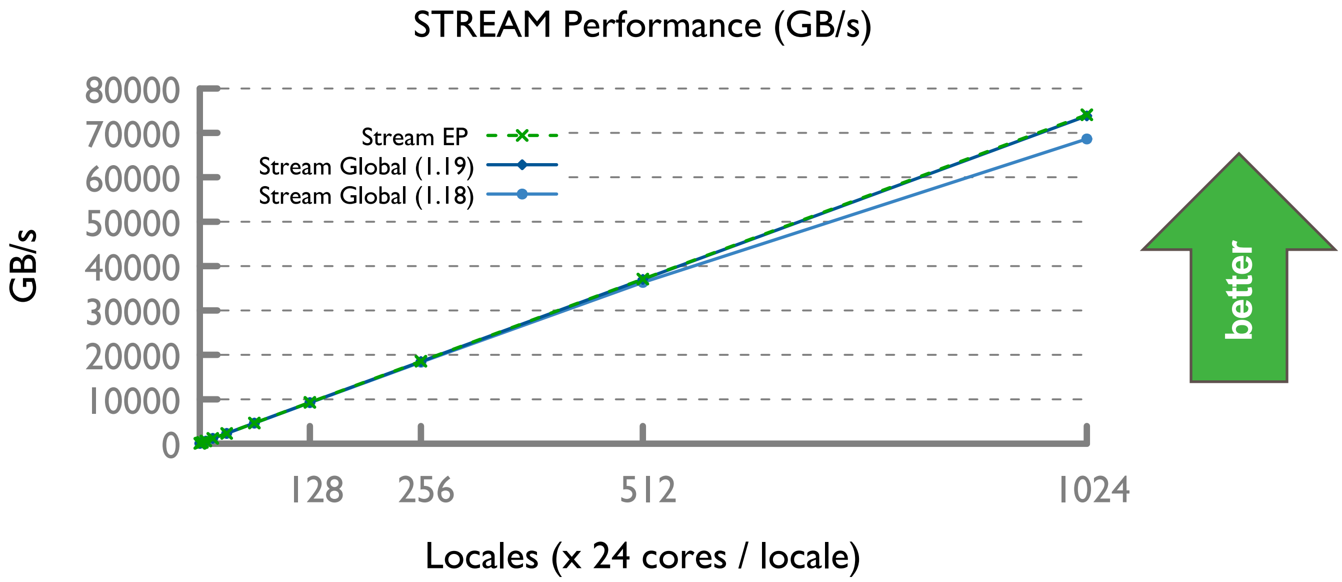
Stream Summary

Stream: Summary

- Optimized iteration over block-distributed arrays
- Made remote-coforall task spawning non-blocking
- Increased size threshold for small remote tasks
    - Most remote tasks can now be initiated with only a single network PUT

Stream: Performance Impact

- Stream Global performance now on par with EP at 1,024 locales



Memory Leaks

Memory Leaks: Background, This Effort

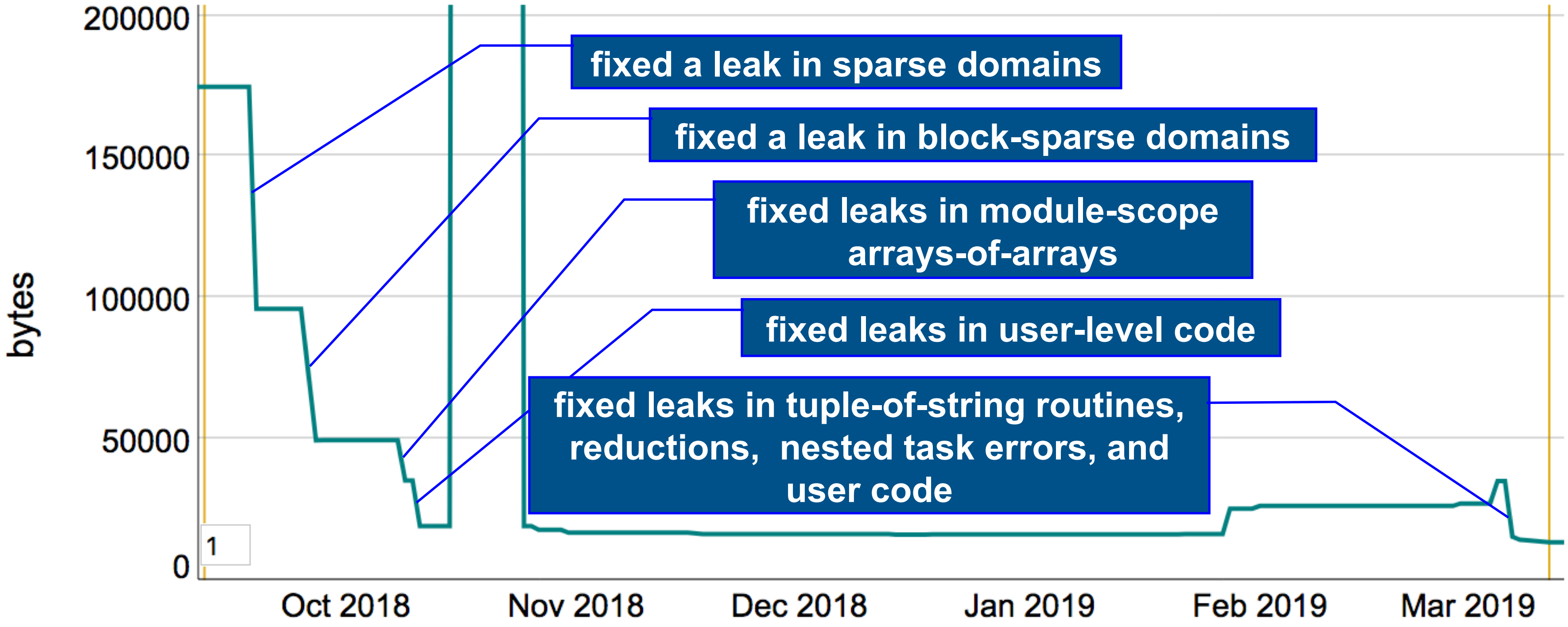
Background:

- Historically, Chapel's test system has leaked a large amount of memory
- Chapel 1.15 and 1.16 closed major sources of large-scale leaks
- Chapel 1.17 and 1.18 closed additional memory leaks

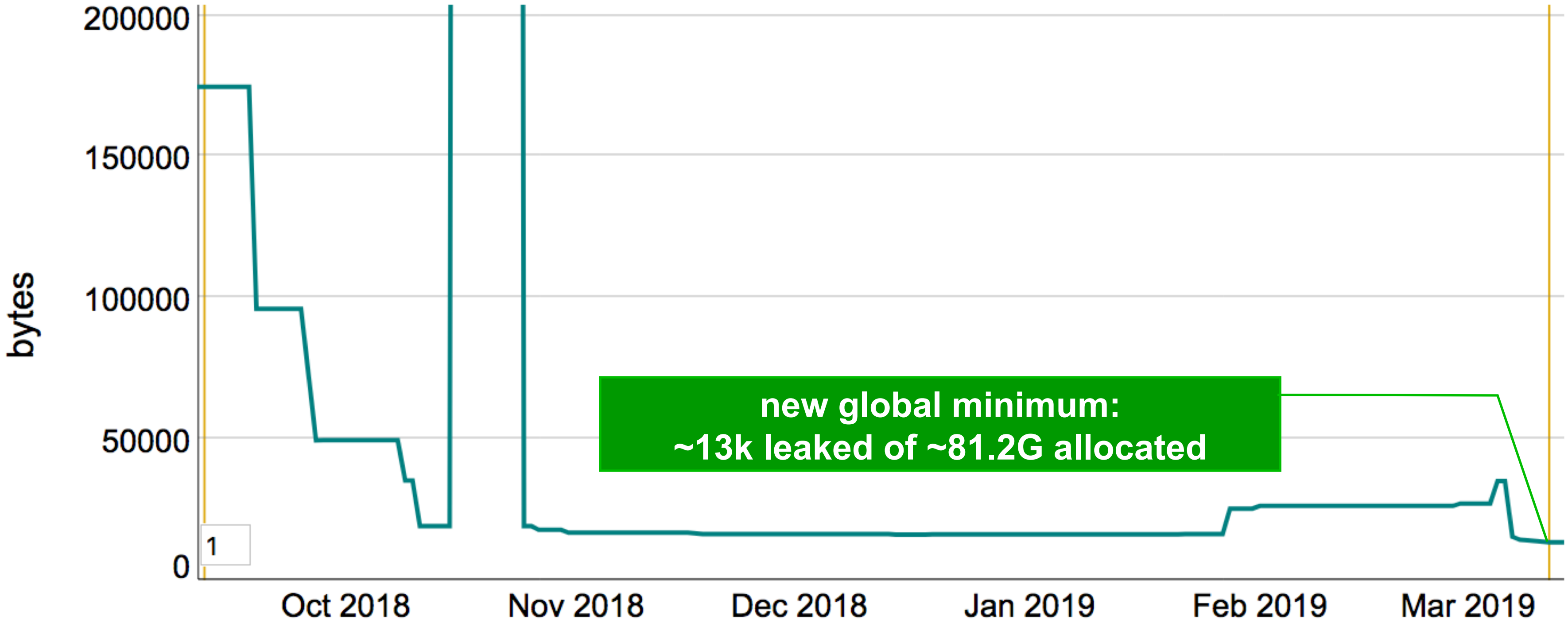
This Effort:

- Closed additional sources of Chapel-introduced leaks, most notably:
  - sparse domains
  - module scope arrays-of-arrays
- Also closed some user-level leaks in the tests themselves

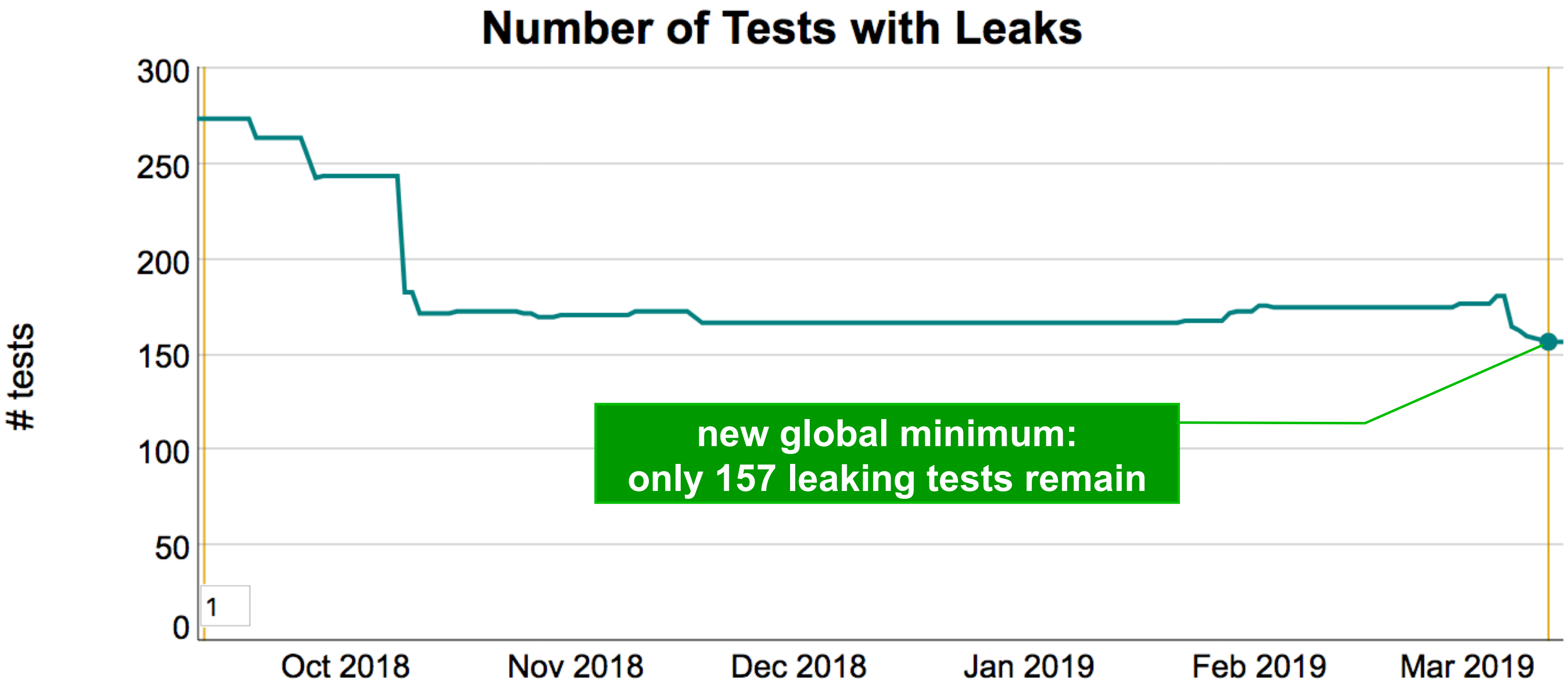
Memory Leaks: This Effort (major fixes)



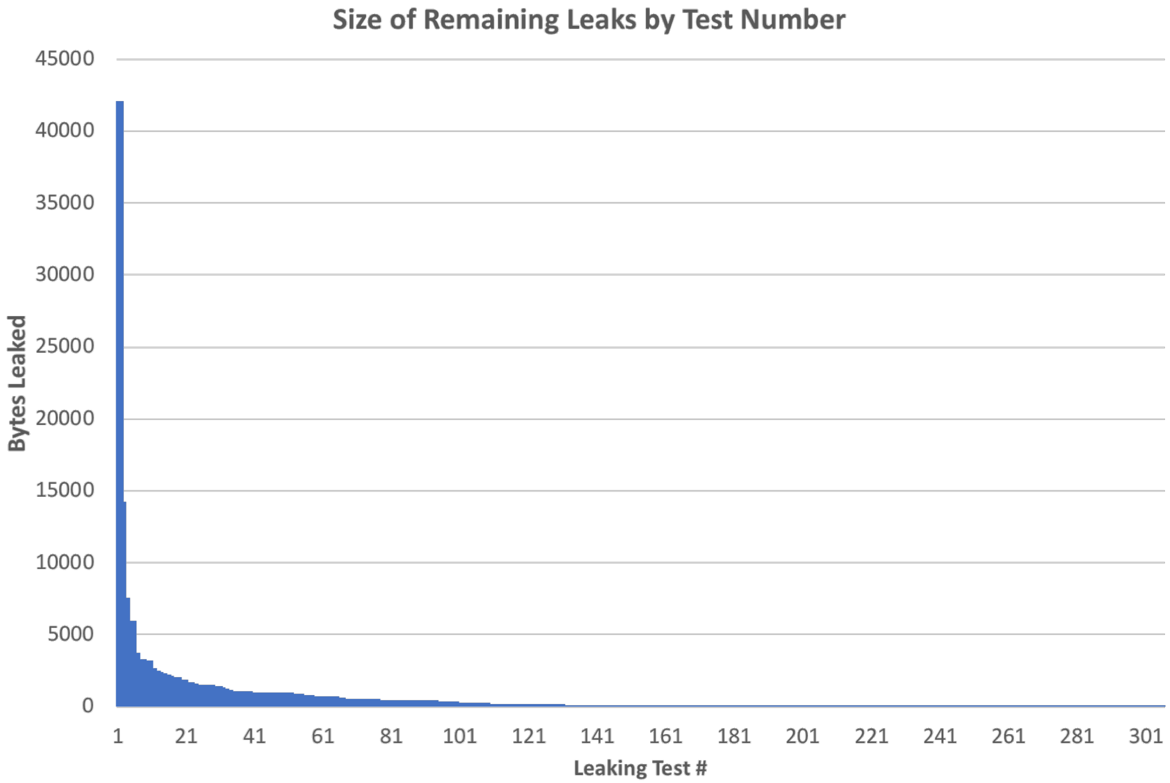
Memory Leaks: Impact



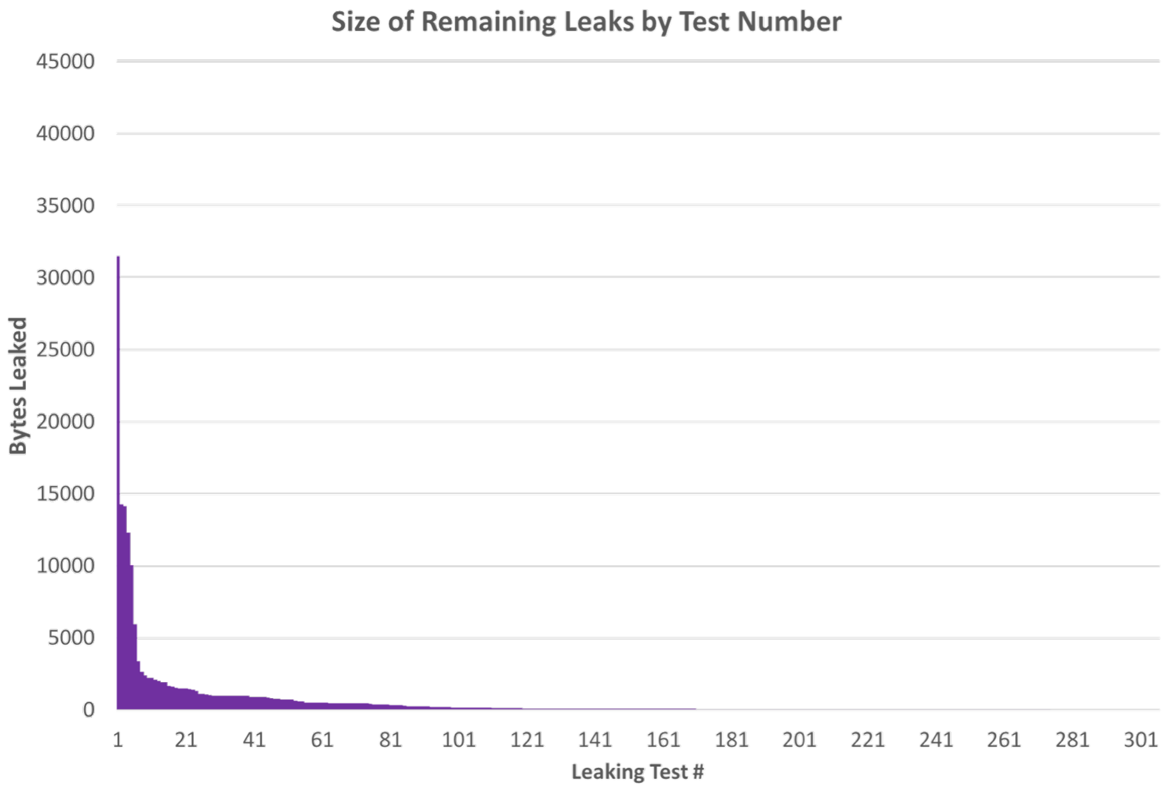
Memory Leaks: Impact



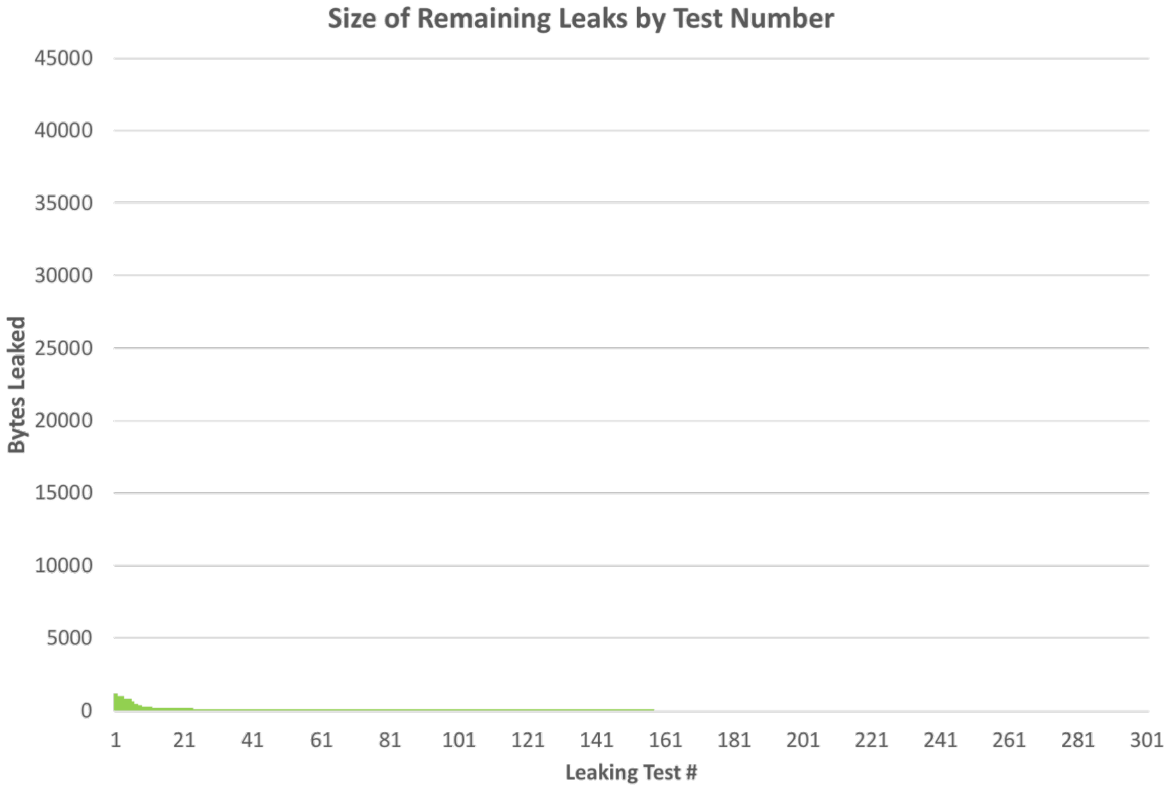
Memory Leaks: Remaining Leaks (as of 1.17)



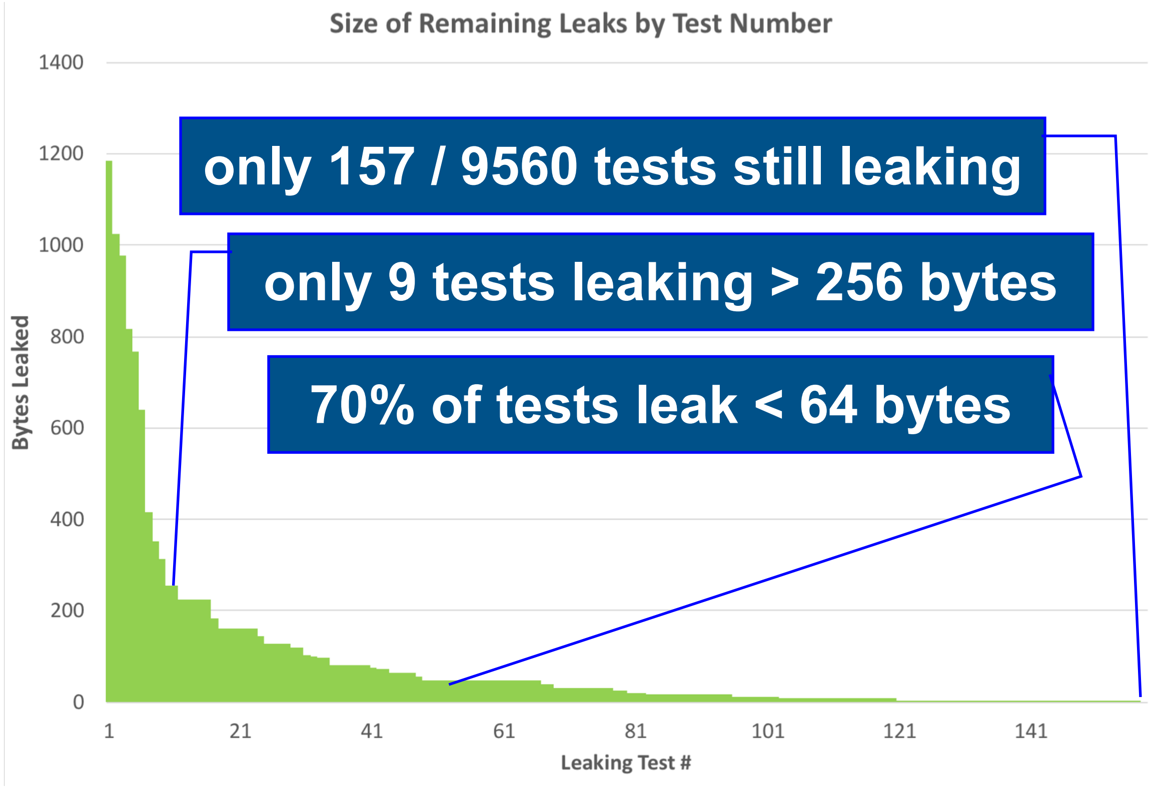
Memory Leaks: Remaining Leaks (as of 1.18)



Memory Leaks: Remaining Leaks (as of 1.19)



Memory Leaks: Remaining Leaks (as of 1.19)



Memory Leaks: Status

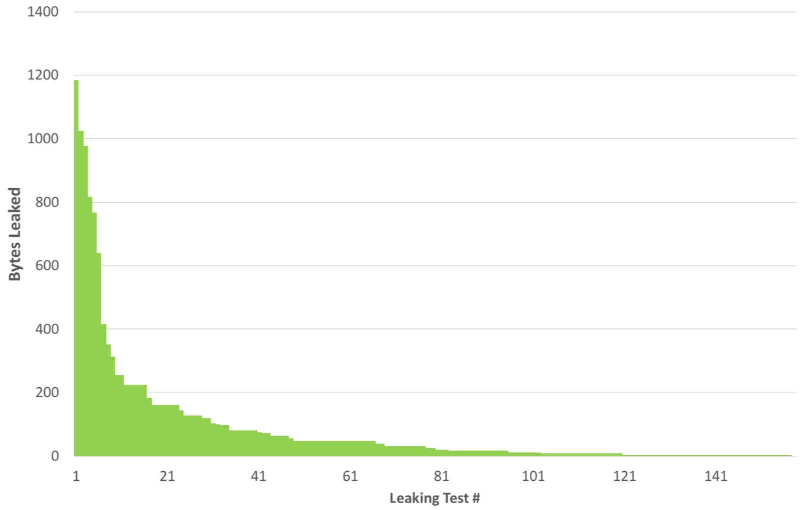
Status:

- From 1.18–1.19, leaks reduced by 92% in testing (w/ ~423 new tests added)
- Primary causes of remaining leaks in testing:
  - user-level leaks
  - certain uses of tuples
  - certain managed class instances
  - certain loop idioms (e.g., breaking out of a non-inlined loop)
  - certain error / defer cases
  - certain runtime type expressions
  - ‘Private’ distributions and ‘Dimensional’ distributions (prototype-grade)
  - first-class functions (not yet officially supported)

Memory Leaks: Next Steps

Next Steps:

- Close remaining leaks
- Cause new leaks to generate errors in nightly testing



For More Information

For a more complete list of related changes in the 1.19 release, refer to the 'Performance Optimizations/Improvements' and 'Memory Improvements' sections in the CHANGES.md file.

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.

THANK YOU

QUESTIONS?

chapel_info@cray.com

@ChapelLanguage

chapel-lang.org

cray.com

@cray_inc

linkedin.com/company/cray-inc-