Benchmarks and Performance Optimizations

Chapel versions 1.21 / 1.22

April 9 / 16, 2020

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

a Hewlett Packard Enterprise company

Outline

- [Array Optimizations]
  - Distributed Array/Domain Creation
  - Fast-Follower Improvements
  - Parallelize PrivateDist Scan
  - Resize Arrays In-Place
- Runtime Improvements
  - InfiniBand Improvements
  - Misaligned GET Improvements
  - Remote Cache Improvements
  - Serial I/O Optimization
- Unordered Operations
  - Unordered Copy Improvements
  - Automatic Unordered Copy
- Memory Leak Improvements

Array Optimizations

Distributed Array/Domain Creation

Array/Domain Creation: Background, This Effort

Background:

- Typically, array/domain creation is not part of a benchmark's timed region
  - However, use cases like Arkouda create many arrays/domains on the fly
- Creation involves a lot of all-to-all communication
  - Previously, this was fine-grained and had significant overhead at scale

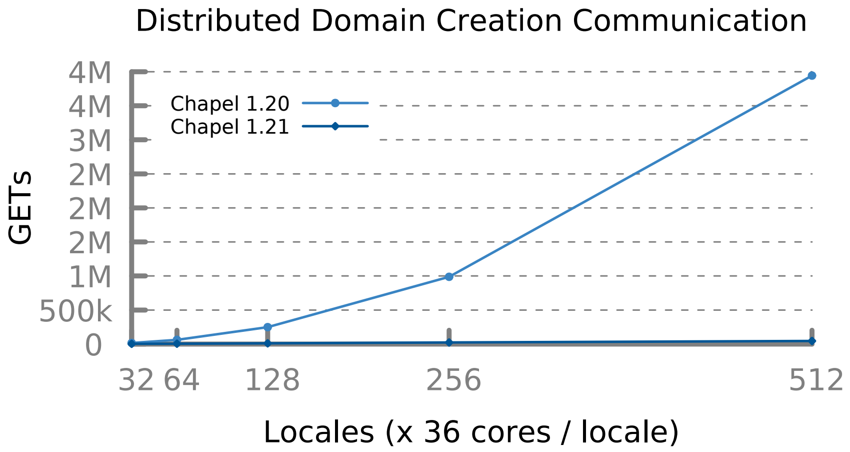
This Effort:

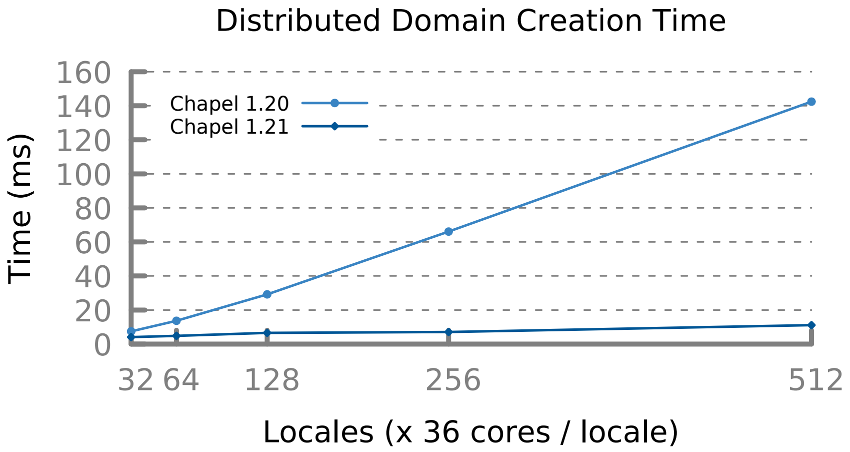
- Use bulk communication distributed during domain and array creation

```
var dom = {1..n} dmapped Block({1..n});    // domain creation
var arr: [dom] real;                       // array creation
```

Domain Creation: Impact

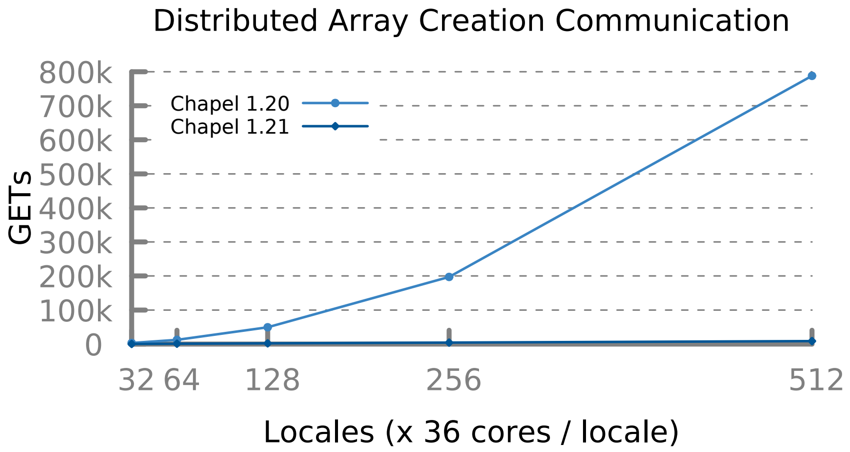
- Significantly faster and more scalable distributed domain creation
  - At 512 locales: 85x fewer GETs, 10x faster

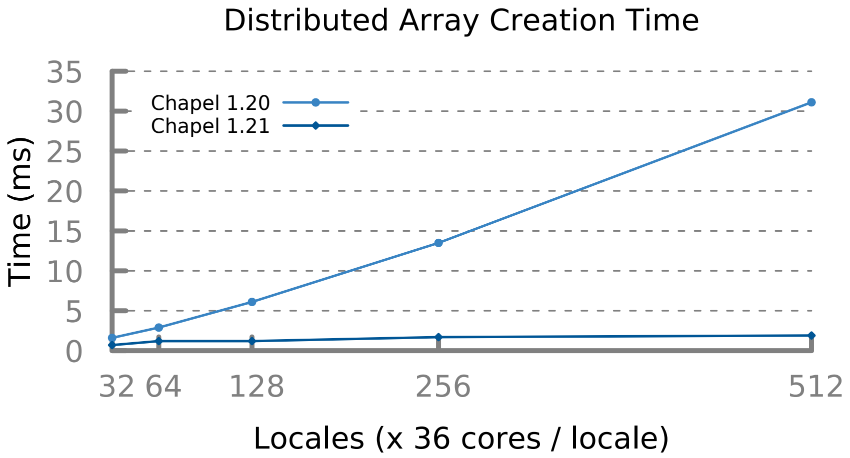




Array Creation: Impact

- Significantly faster and more scalable distributed array creation
  - At 512 locales: 90x fewer GETs, 15x faster





Fast-Follower Improvements

Fast-Follower: Background, This Effort

Background: "Fast Followers" optimize iteration over aligned distributed arrays

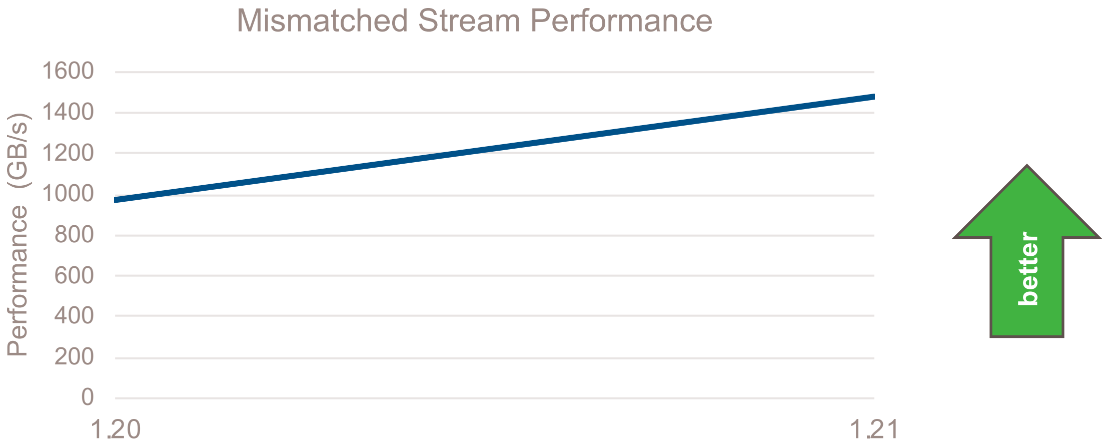
- Can skip locality checks when arrays have the same distribution/alignment
- Previously, arrays had to have identical domains and element types

```
var A, B = newBlockArr({1..n}, real);
var    C = newBlockArr({1..n}, int);
A = B + 3.0 * C; // not optimized, different types, not identical domains
```

This Effort: Enable optimization for equivalent domains and any element type

Fast-Follower: Impact

- Fast followers now trigger in more cases
  - 1.5x improvement for Stream with different element types



Parallelize PrivateDist Scan

PrivateDist Scan: Background, This Effort

Background: PrivateDist provides replicated values across all locales

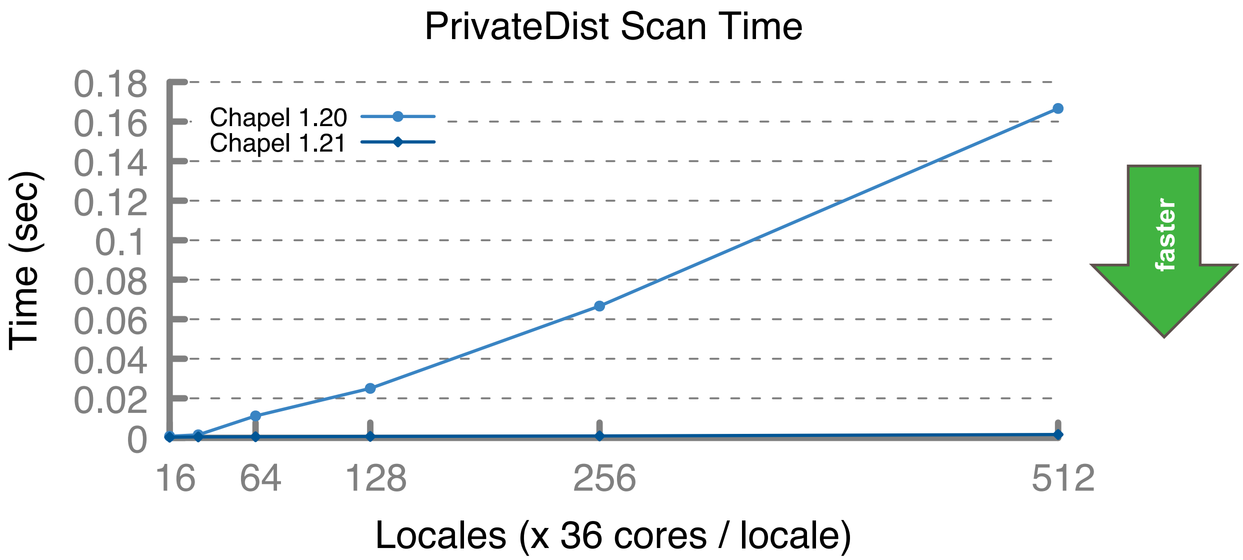
- Scans over PrivateDist arrays were serialized
- Resulted in poor scalability and a compile-time serialization warning

This Effort: Parallelized scans over PrivateDist arrays

- Eliminated serialization warning

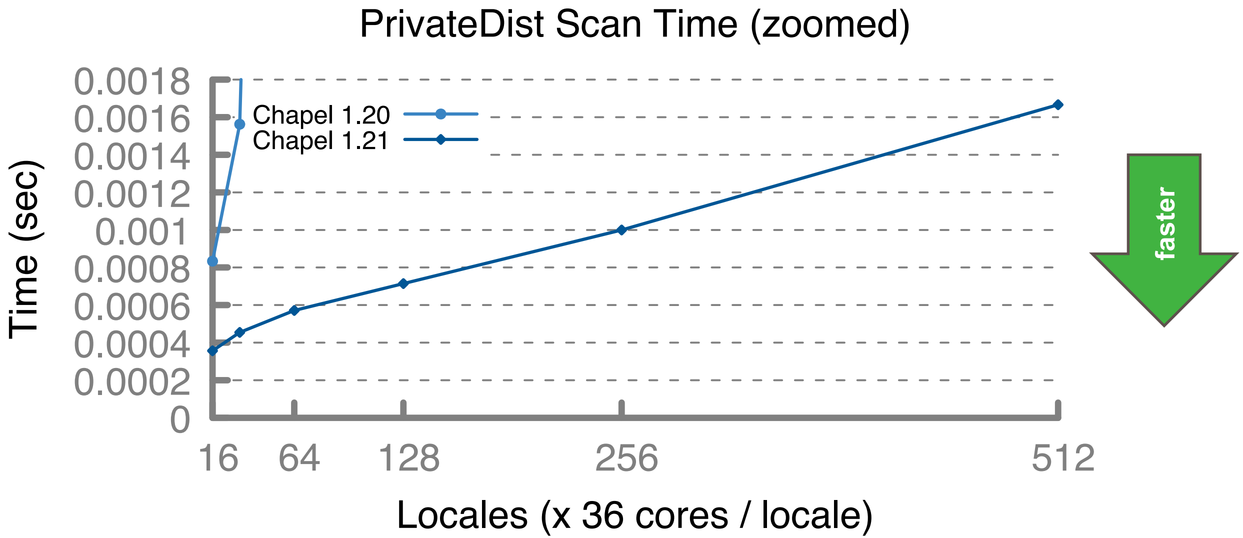
PrivateDist Scan: Impact

- Significantly improved scan performance and scalability
  - 100x speedup at 512 nodes



PrivateDist Scan: Impact

- Significantly improved scan performance and scalability
  - 100x speedup at 512 nodes



Resize Arrays In-Place

Resize-in-Place: Background

- Chapel reallocates arrays based on indices, not memory layout

```
var D = {1..3};
var A: [D] int = [1, 2, 3];
D = {0..4};
writeln(A);    // prints "0, 1, 2, 3, 0", not "1, 2, 3, 0, 0"
```

- For this reason, rectangular arrays have traditionally been resized by:
  1. allocating a new array
  2. copying over elements within the intersection of D_{old} and D_{new}
  3. ensuring that any new elements are default-initialized

Resize-in-Place: This Effort

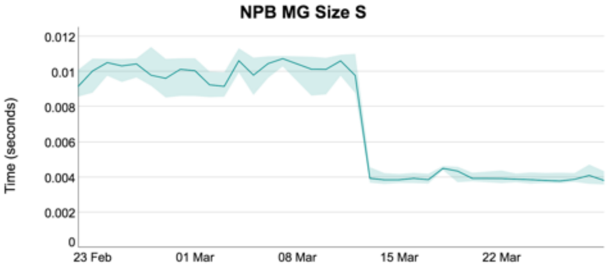
- However, some array resizings are amenable to being done in-place
  - Notably, 1D array resizings in which the low bound and stride don't change:

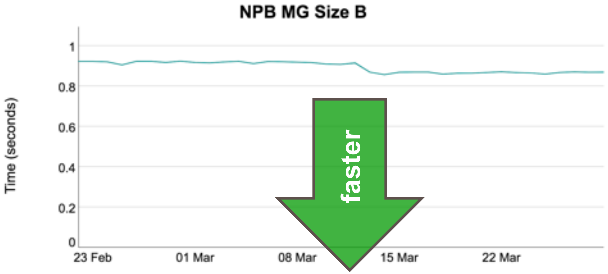
```
var D = {1..3};
var A: [D] int = [1, 2, 3];
D = {1..5};   // D's prefix & stride are identical, so we need not move the array
writeln(A);   // prints "1, 2, 3, 0, 0"
```

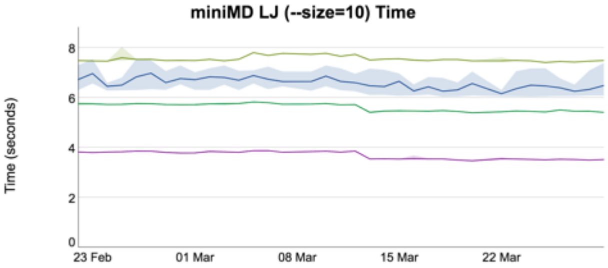
- As a result, such cases are now handled by:
  1. calling realloc() on the array's buffer
  2. ensuring that any new elements are default-initialized
- Note: realloc() can't always resize in place; but when it can, there's a benefit

Resize-in-Place: Impact

- CLBG reverse-complement improved by ~7.7% on UMA compute nodes
  - No significant impact on NUMA compute nodes
    - (challenging to do anything smart w.r.t. NUMA locality when resizing)
- Also resulted in improvements for a few other benchmarks:







Resize-in-Place: Status and Next Steps

Status:

- A minor improvement, yet one that does no harm and is nice when it helps

Next Steps:

- As motivating examples arise:
  - extend realloc-in-place to other cases (e.g., multidimensional arrays)
  - tune implementation further for NUMA compute nodes

Runtime Improvements

InfiniBand Improvements

InfiniBand: Background, This Effort

Background: Started running InfiniBand performance testing last release

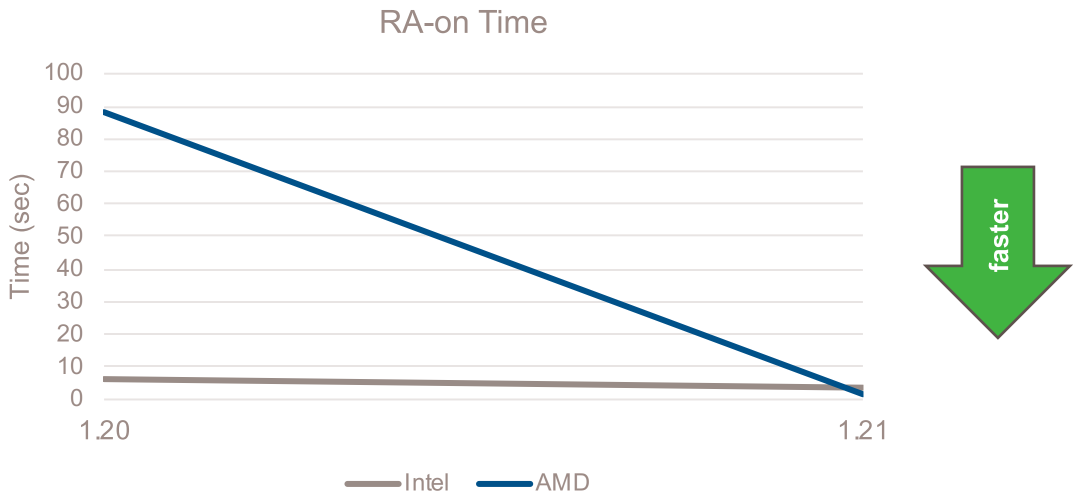
- Nightly configuration runs on machine with Intel processors
- Ran some experiments on AMD EPYC processors
  - Saw significant performance degradations for ‘on’-heavy workloads

This Effort: Optimized on-statements for InfiniBand

- Serialized calls to on-statement handlers to reduce contention

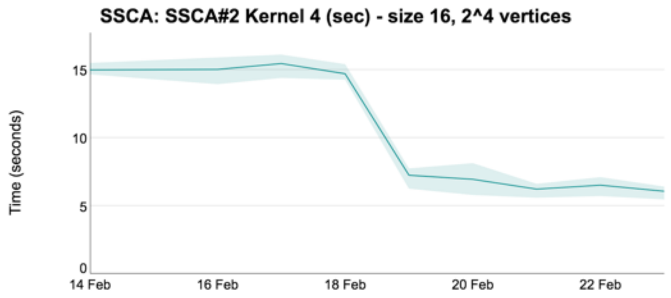
InfiniBand: Impact

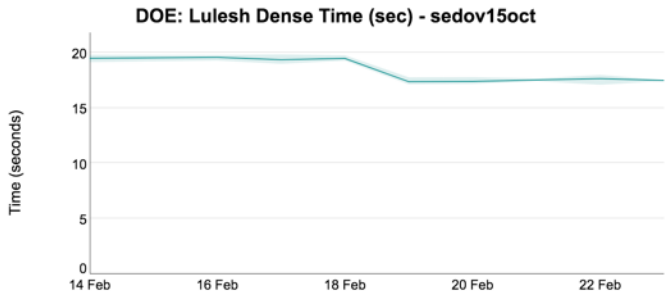
- Significantly improved on-statement performance
  - For RA-on using 48-core Intel Cascade Lake and AMD Rome processors
    - 2x speedup on Intel, 55x speedup on AMD



InfiniBand: Impact, Next Steps

Impact: Improvements for other on-statement-heavy benchmarks





Next Steps: Better understand root cause of original degradation

- Unknown why performance was more impacted on AMD processors

Misaligned GET Improvements

Misaligned GET: Background, This Effort

Background: GETs on the Aries NIC must be 4-byte aligned

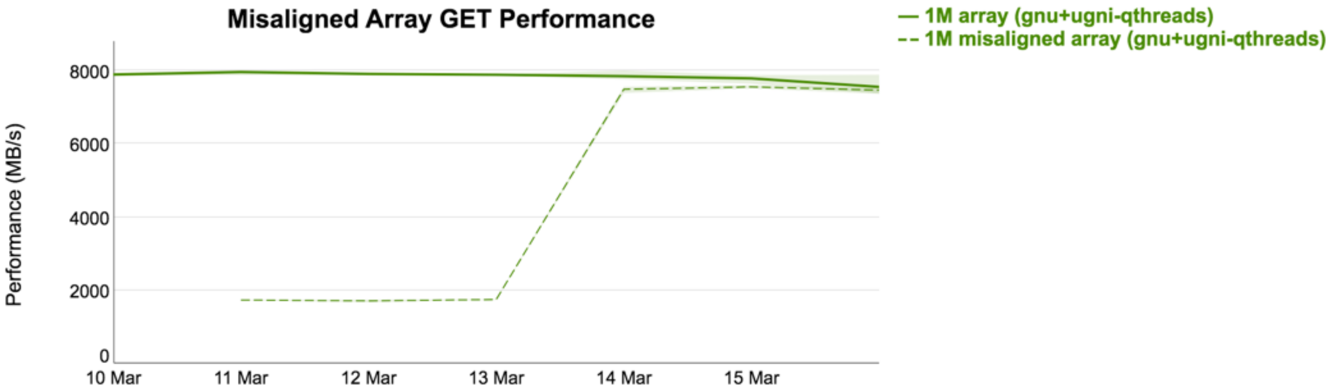
- When not aligned, perform GET to bounce buffer, copy to target buffer
- Previously, this code path was not well-tested
  - Most Chapel benchmarks use 8-byte int(64)/real(64)
  - Arkouda uses uint(8)/bool arrays, which exposed several bugs

This Effort: Improve misaligned GETs

- Fixed correctness issues and added additional tests
- Optimized large misaligned GETs
  - Use 0-copy GET for aligned interior; only bounce misaligned head/tail

Misaligned GET: Impact

- Improved performance for large misaligned GETs
  - 4x improvement for transfers larger than 1MB



Remote Cache Improvements

Remote Cache: Background

- Chapel has a cache for remote data that can be enabled with --cache-remote
  - Can provide significant speedups for suboptimal communication patterns
  - Supports read-ahead and write-behind, can eliminate repeated communication

```
var A, B:[1..n] int;
on Locales[1] do
  for i in 1..n do
    B[i] = A[i];

//  A[i] normally 8-byte GET, done in 1024-byte chunks with cache read-ahead
// B[i] normally 8-byte PUT, done in 1024-byte chunks with cache write-behind
// Normally repeated GETs for array metadata, only 1 GET with cache
```

Remote Cache: Background

- Can provide large speedups for real workloads, especially on slower networks
  - 3x speedup for MiniMD on Aries, 100x on FDR InfiniBand
  - 20x speedup for PTRANS on Aries, 500x on FDR InfiniBand

- Previously, there were several issues that prevented it from being recommended
  - Large performance regressions for some workloads
  - Hangs or crashes under comm=ugni
  - Not regularly tested

Remote Cache: This Effort

- Addressed --cache-remote correctness issues
  - Fixed several hangs and crashes under ugni
  - Fixed support for unorderedCopy
  - Fixed support for guard pages
- Eliminated known performance overheads
  - Bypass cache for large transfers
- Added nightly testing across all communication implementations

Remote Cache: Status, Next Steps

Status: --cache-remote is stable enough to recommend to users

- Can provide substantial improvements for suboptimal communication patterns

Next Steps: Explore enabling --cache-remote by default

- Want to explore synthetic benchmarks to better tune
- Need to evaluate performance and memory overhead at scale

Serial I/O Optimization

Serial I/O: Background, This Effort

Background: Chapel I/O is parallel-safe by default

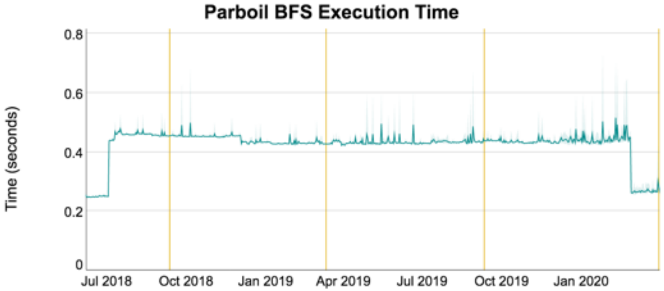
- 1.18 changed I/O to use a sync lock to improve parallel performance
- However, this hurt the performance of serial I/O

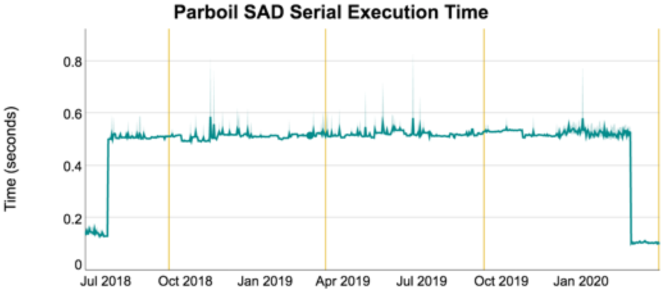
This Effort: Switched to an optimized atomic spinlock for I/O

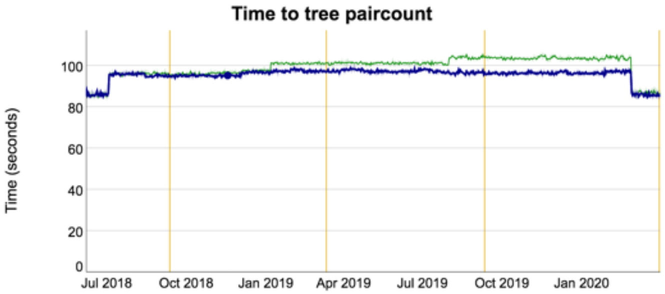
- Has minimal serial overhead while maintaining good parallel performance

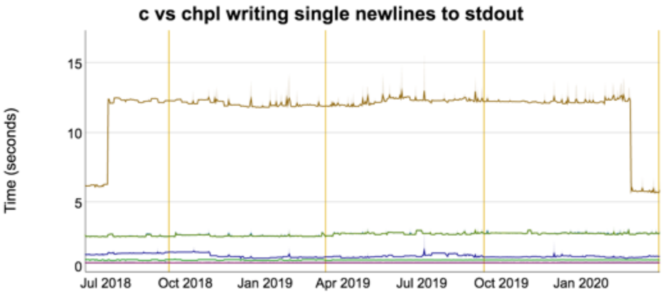
Serial I/O: Impact

- Resolved previous serial I/O regressions









Unordered Operations

Unordered Copy Improvements

Unordered Copy: Background, This Effort

Background: 'unorderedCopy(dst, src)' is a faster, non-sequential consistent copy

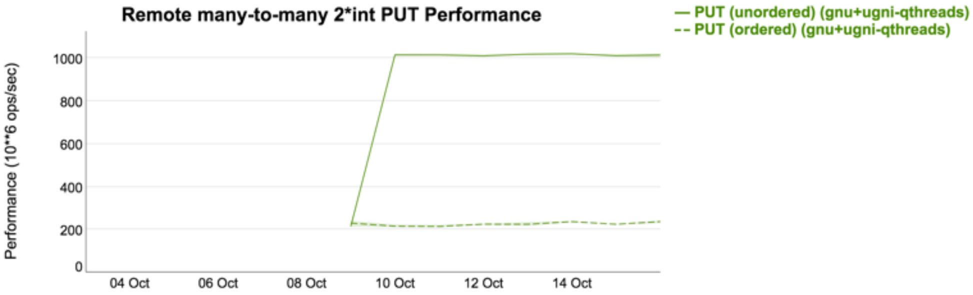
- Previously, it was only implemented for numeric and bool types

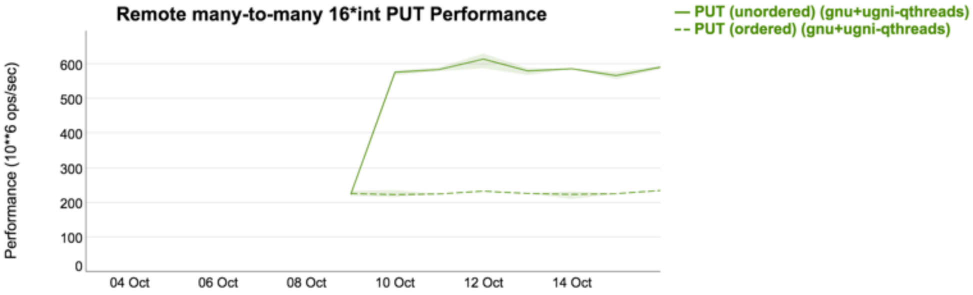
This Effort: Extended support to all trivially copyable types

- numeric/bool
- numeric/bool tuples
- numeric/bool records with no copy-init, deinit, or assignment overload

Unordered Copy: Impact

- Faster copies for trivially copyable types
  - 4.5x speedup for 2*int tuple, 2.5x for 16*int





Automatic Unordered Copy

Auto Unordered Copy: Background, This Effort

Background: Unordered operations provide a significant performance speedup

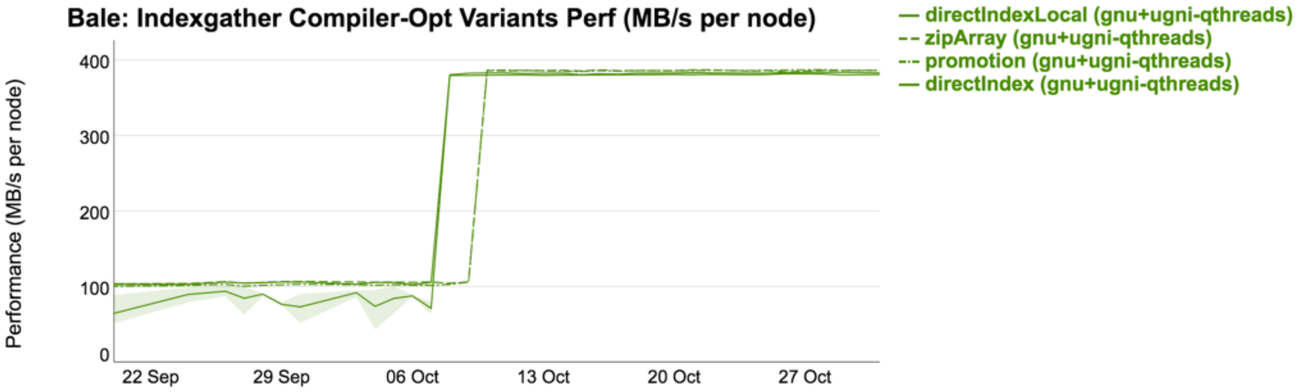
- But they are an advanced feature that break the memory consistency model
- 1.20 enabled an optimization to automatically use unordered ops
  - Triggered for array indexing (e.g. 'A[i]'), but not other iteration idioms

This Effort: Extended compiler optimization

- Handles promotion, zippered iteration, direct array iteration

Auto Unordered Copy: Impact

- More idioms can be optimized by compiler
  - Bale indexgather variants automatically optimized



Memory Leak Improvements

Memory Leaks: Background, This Effort

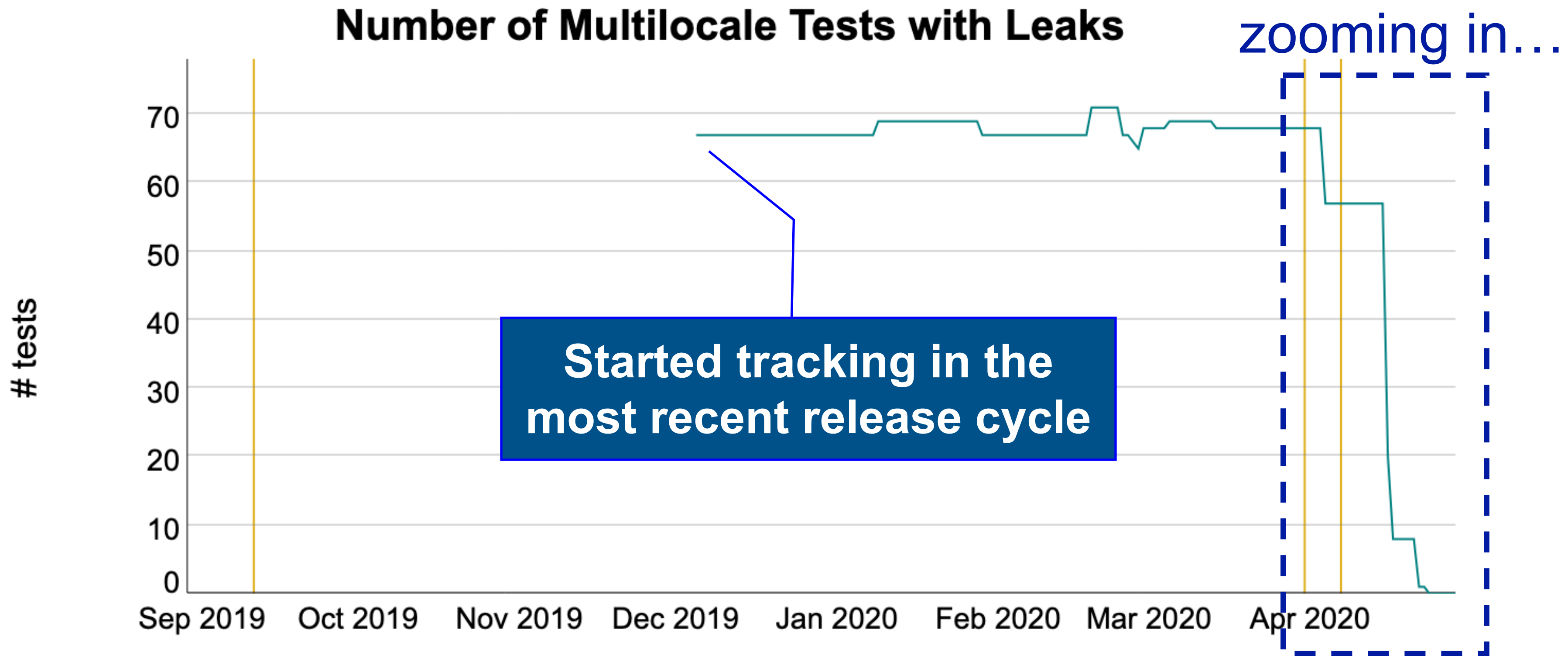
Background:

- Recent releases closed many major memory leaks
  - However, we monitored only single-locale leaks
- A few important configurations were not tested for memory leaks
  - e.g. multi-locale, LLVM backend

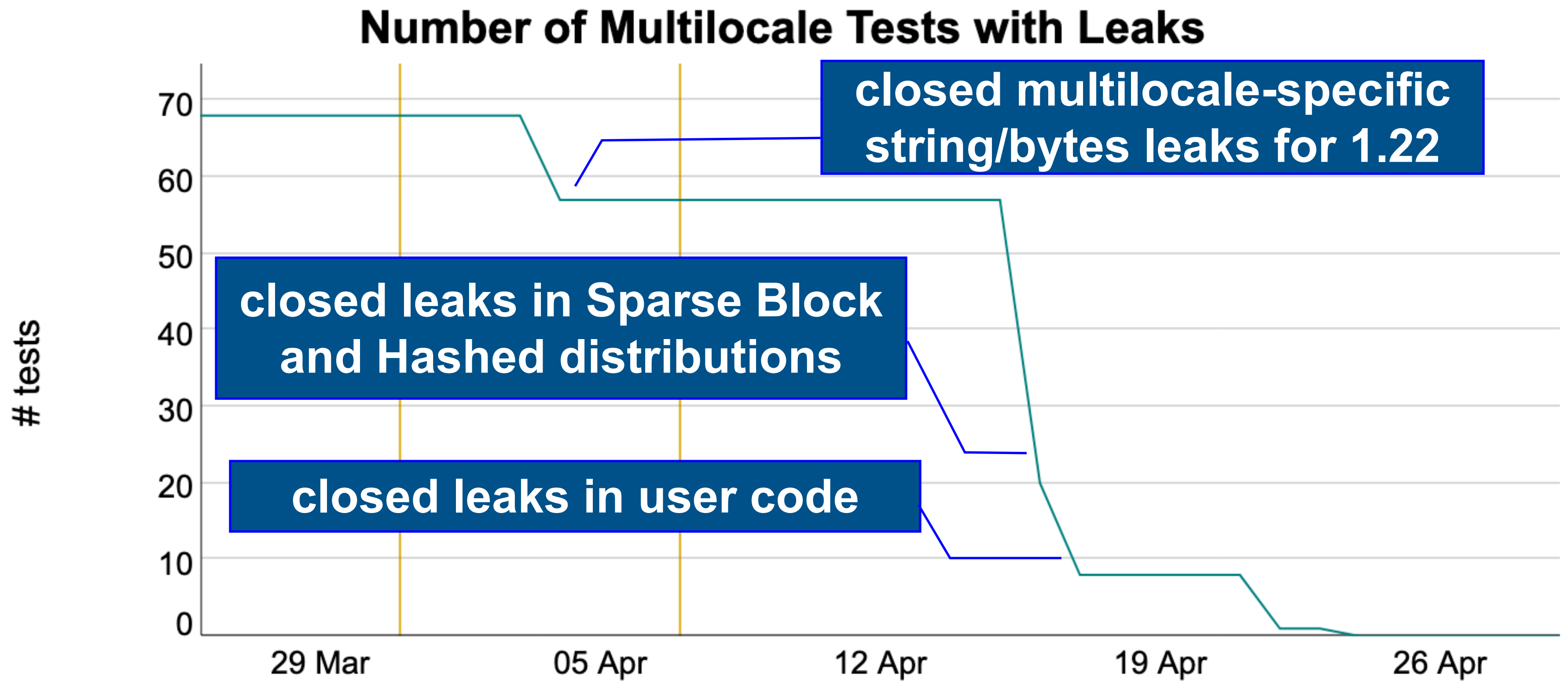
This Effort:

- Track and close multi-locale leaks
  - Different set of leaks that we do not catch in single-locale runs
- Verified that there are no leaks specific to using LLVM back-end
- Reduction in single-locale leaks

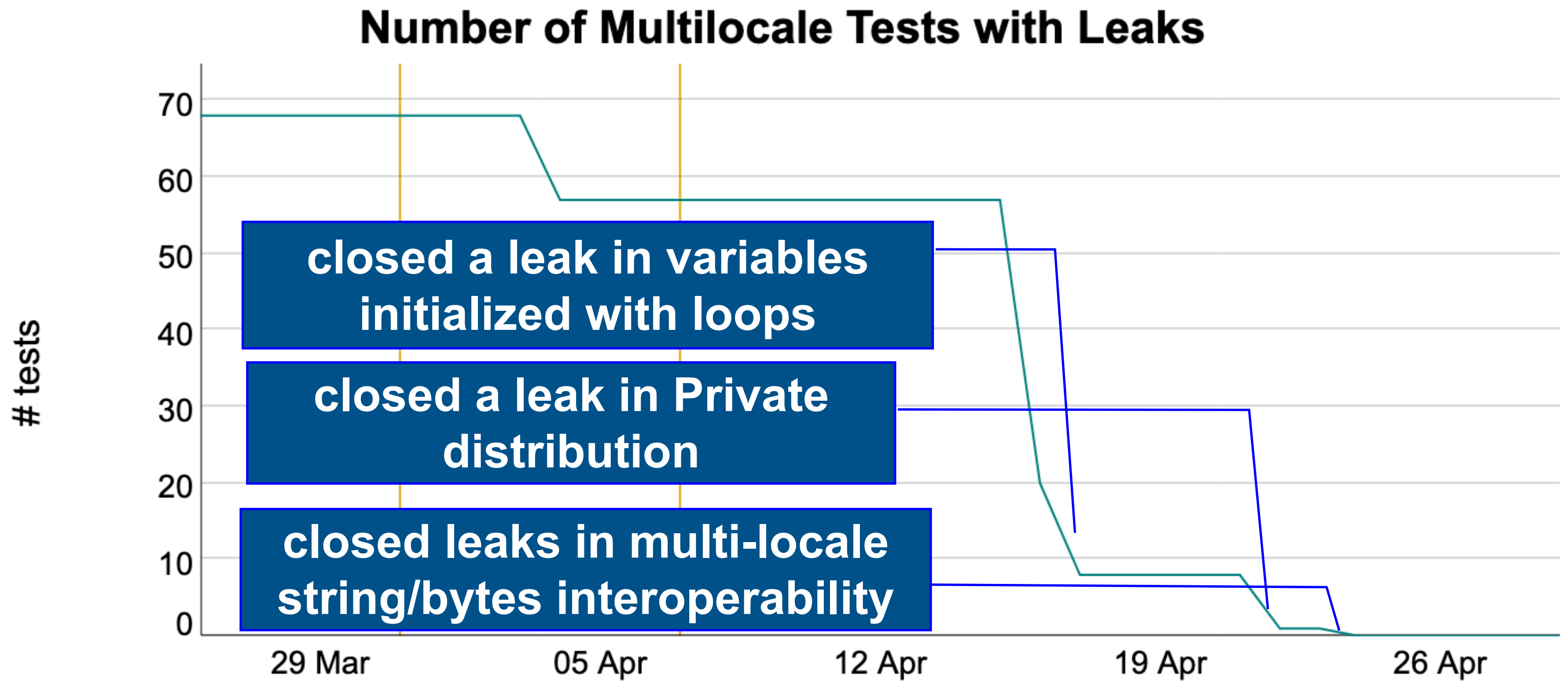
Memory Leaks: Multi-locale Leaks



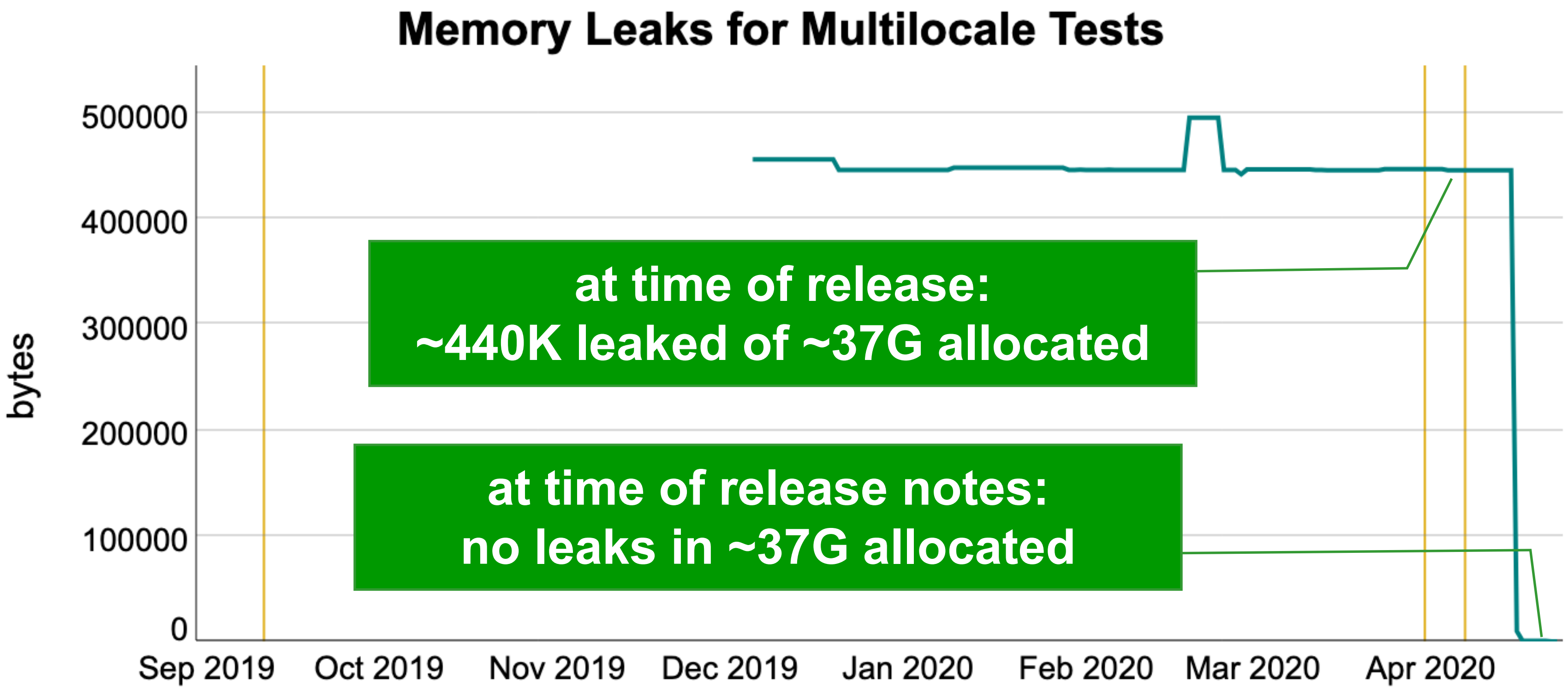
Memory Leaks: Multi-locale Leaks (zoomed)



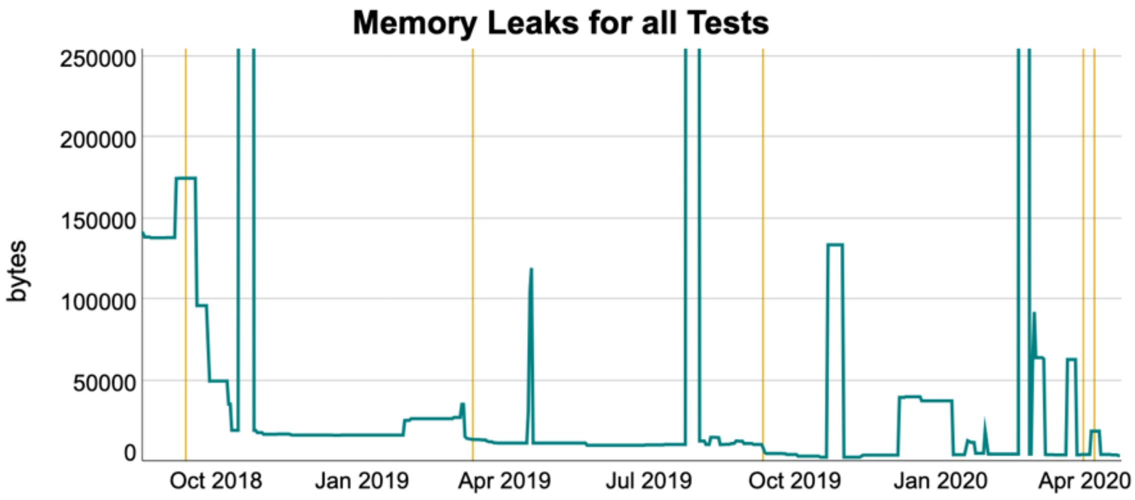
Memory Leaks: Multi-locale Leaks (zoomed)



Memory Leaks: Multi-locale Leaks (volume)



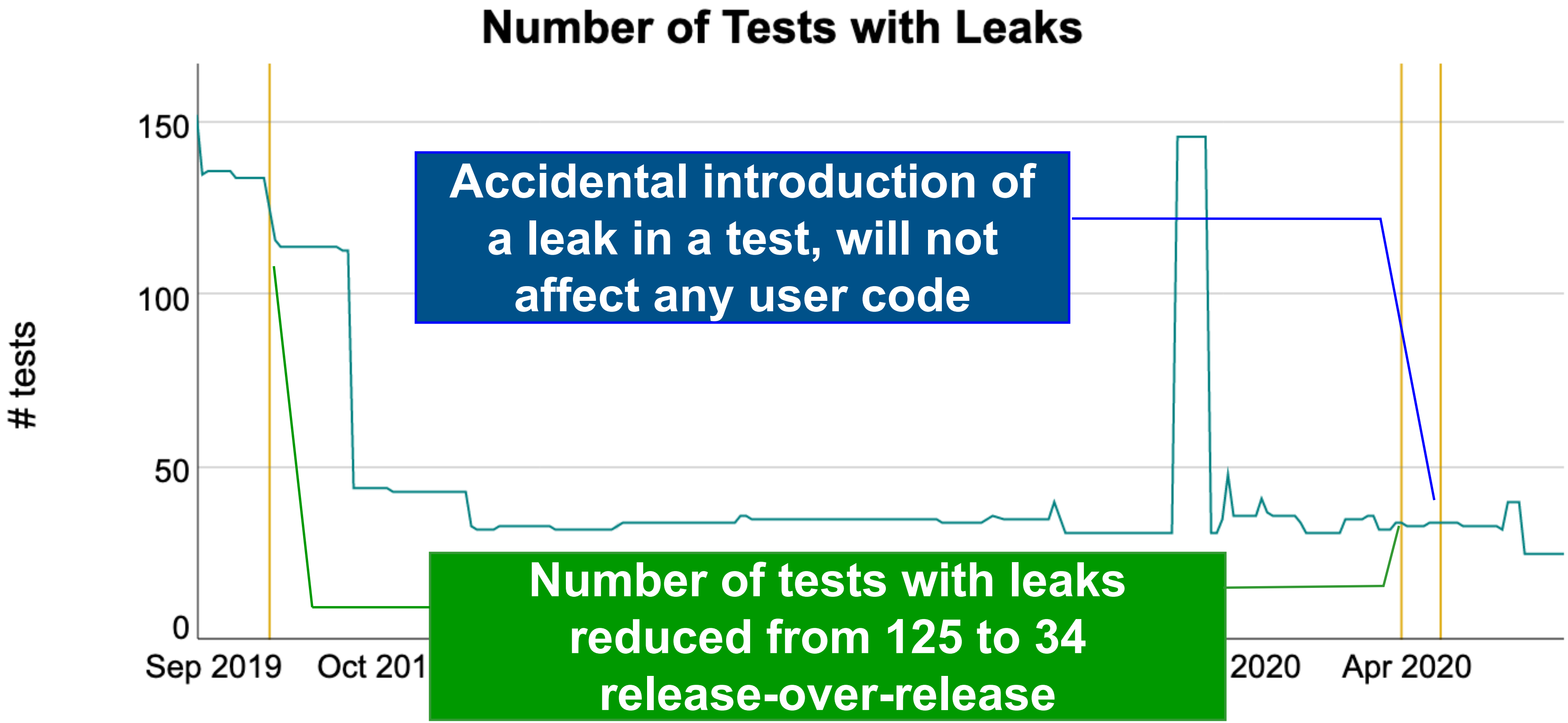
Memory Leaks: Single-locale Leaks



very active release cycle in terms of memory changes

- Compiler changes
  - Split initialization
  - Copy elision
  - Early deinitizalization
- Test changes
  - Start tracking mason

Memory Leaks: Single-locale Leaks



Memory Leaks: Impact, Next Steps

Impact:

- Closed all known multi-locale memory leaks
- Reduced single-locale memory leaks
- Confirmed that there is no additional leak when using LLVM backend

Next Steps:

- Eliminate remaining single-locale leaks
- Make new leaks a correctness error

For More Information

For a more complete list of related changes in the 1.21 and 1.22 releases, refer to the 'Performance Improvements' and 'Memory Improvements' sections in the CHANGES.md file

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.

THANK YOU

QUESTIONS?

chapel_info@cray.com
@ChapelLanguage
chapel-lang.org

CRAY
a Hewlett Packard Enterprise company

cray.com
@cray_inc
linkedin.com/company/cray-inc-/