CHAPEL 1.23 RELEASE NOTES: PERFORMANCE OPTIMIZATIONS

Chapel Team

October 15, 2020

OUTLINE

- Array Optimizations
- Compilation Time Improvements
- Memory Improvements

ARRAY OPTIMIZATIONS

- Automatic Local Access Optimization
- Improvements to Associative Types
- Array Tracking Optimization
- Constant Domain Optimization
- Parallel Array Initialization
- Parallel Array Assignment
- Array Swap Optimization

AUTOMATIC LOCAL ACCESS OPTIMIZATION

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Background

- Iterating over arrays/domains using 'forall' is a very common pattern in Chapel:

```
var D = newBlockDom({1..N});
var A: [D] int;
forall i in D do
  A[i] = calculate(i);
```

loop is run over the domain of an array

the array is indexed using the loop index

- For distributed arrays, every 'A[i]' checks whether it is a local access
  - This check is overhead for this pattern: they are all guaranteed to be local
- Potential workarounds:

```
forall (a, i) in zip(A, A.domain) do
  a = calculate(i);

forall i in A.domain do
  A.localAccess(i) = calculate(i);
```

clunky

AUTOMATIC LOCAL ACCESS OPTIMIZATION

This Effort

- Implemented a compiler analysis that replaces 'A[i]' with 'A.localAccess[i]'
  - The optimization is done statically if the compiler can prove that:
    - the loop domain supports the optimization
    - the array is indexed with the loop index symbol
    - the loop domain matches the array's domain
  - The optimization is subject to a dynamic check at execution time if:
    - the first two conditions above are met, but the compiler cannot prove that the loop and array domains match

- An example where the optimization can be done statically:

```
var D = newBlockDom({1..10});
var A: [D] int;
forall i in D do
  A[i] = calculate(i);   // ==>  A.localAccess[i] = calculate(i);
```

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Arrays With Common Domains

- The optimization also applies to multiple arrays

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;
forall i in D do
  A[i] = calculate(B[i]);
```

array(s) indexed using the loop index

loop is run over the domain of array(s)

- Even when the loop domain is not explicit

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;
forall i in A.domain do
  A[i] = calculate(B[i]);
```

array(s) indexed using the loop index

loop is run over a domain query

array(s) have the same domain as the loop

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Dynamic Checks

- If the compiler cannot determine the domain of an array:
  - Equality of domains will be checked at execution time
  - Depending on that, an optimized or unoptimized version of the loop will be run

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);    // currently we can't infer 'B' has the same domain as 'A'
forall i in A.domain do
  A[i] = calculate(B[i]);    // B[i] is local if A.domain == B.domain
                             // that can only be confirmed at execution time
```

- Terminology
  - 'A' is a static candidate
  - 'B' is a dynamic candidate

- The compiler will clone loops if there are one or more dynamic candidates
  - This can increase compilation time

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Dynamic Checks

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
param staticCheckA = canUseLocalAccess(A, A.domain);
param staticCheckB = canUseLocalAccess(B, A.domain);
if staticCheckA || staticCheckB {
  const dynamicCheckB = canUseLocalAccessDyn(B, A.domain);
  if dynamicCheckB then
    forall i in A.domain do
      A.localAccess[i] = calculate(B.localAccess[i]);
  else
    forall i in A.domain do
      A.localAccess[i] = calculate(B[i]);
} else {
  forall i in A.domain do
    A[i] = calculate(B[i]);
}
```

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
forall i in A.domain do
  A[i] = calculate(B[i]);
```

Static checks are created for both arrays

Dynamic check is created only for B

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Dynamic Checks

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
param staticCheckA = canUseLocalAccess(A, A.domain);
param staticCheckB = canUseLocalAccess(B, A.domain);
if staticCheckA || staticCheckB {
  const dynamicCheckB = canUseLocalAccessDyn(B, A.domain);
  if dynamicCheckB then
    forall i in A.domain do
      A.localAccess[i] = calculate(B.localAccess[i]);
  else
    forall i in A.domain do
      A.localAccess[i] = calculate(B[i]);
} else {
  forall i in A.domain do
    A[i] = calculate(B[i]);
}
```

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
forall i in A.domain do
  A[i] = calculate(B[i]);
```

Will be executed if

- A passes static checks
- B passes static and dynamic checks

Will be executed if

- A passes static checks
- B fails static or dynamic checks

Will be executed if

- Neither array passes static checks

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Dynamic Support for Subset Domains

- The optimization covers cases where the loop domain is a subset of the array domain

```
var D = newBlockDom({1..10});
var A, B: [D] int;
forall i in D.expand(-1) do
  A[i] = calculate(B[i]);
```

Optimized upon a dynamic check

- It also detects iteration over (a subset of) the local subdomain of a distributed array's domain

```
var D = newBlockDom({1..10});
var A, B: [D] int;
coforall l in Locales do on l {
  forall i in D.localSubdomain() do
    A[i] = calculate(B[i]);
  // ... or ...
  forall i in D.localSubdomain().expand(-1) do
    A[i] = calculate(B[i]);
}
```

Optimized upon a dynamic check

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Queried Domains in Array Formals

- Static optimization opportunities for array formals without domain queries are limited

```
proc foo(A, B) {
  forall i in A.domain do
    A[i] = calculate(B[i]);
}
```

'A[i]' can be optimized statically

Currently, we can't determine whether B is an array early enough during compilation, so we use dynamic checks for it

- To avoid dynamic checks and loop cloning, be more explicit when multiple arguments share a domain

```
proc foo(A: [?D], B: [D]) {
  forall i in A.domain do
    A[i] = calculate(B[i]);
}
```

We know that B is an array that has the same domain as the loop domain

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Available Compiler Flags

- --[no-]auto-local-access
  - Enable/disable this optimization
  - Enabled by default

- --[no-]dynamic-auto-local-access
  - Enable/disable dynamic optimization
  - Enabled by default
  - Dynamic optimization results in loop cloning and can increase compilation time in some codes

- --[no-]report-auto-local-access
  - Enable/disable verbose output about the optimization steps
  - Disabled by default

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Caveats

- The optimization is thwarted if
  - The locale changes between the 'forall' and the array access

```
forall i in A.domain do
  on Locales[X] do          // this statement can move the execution to another locale
    A[i] = calculate(i);
```

  - The array index symbol is not identical to the loop index symbol

```
forall i in A.domain {
  const k = i;
  A[k] = calculate(i);
}
```

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Caveats

- Zippered foralls are supported only if the loop index is expanded

```
forall (i,a) in zip(D, someIterator()) { }    // the loop will be analyzed further

forall idx in zip(D, someIterator()) { }      // the loop will not be analyzed further
```

- Indexing into shadow variables is not analyzed

```
forall i in D with (ref A) do
  A[i] = calculate(i);
```

- Indexing into array views is not analyzed

```
var A = otherArr[2..10];
forall i in A.domain do
  A[i] = calculated(i);
```

AUTOMATIC LOCAL ACCESS OPTIMIZATION

Impact

- Global STREAM with array indexing:

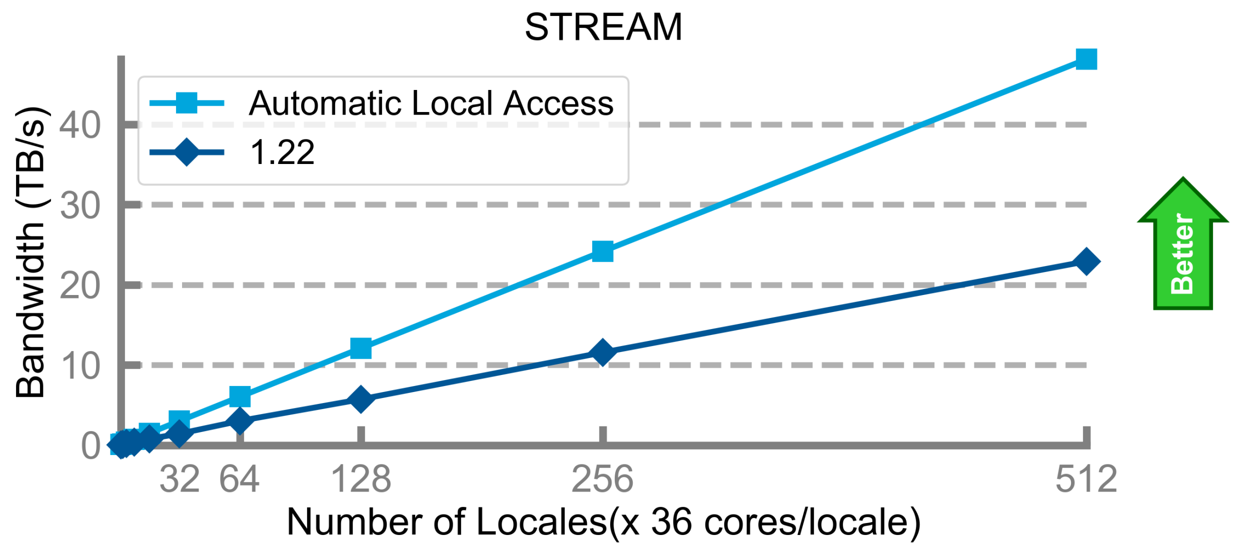
```
forall i in ProblemSpace do
  A[i] = B[i]+ alpha * C[i];
```

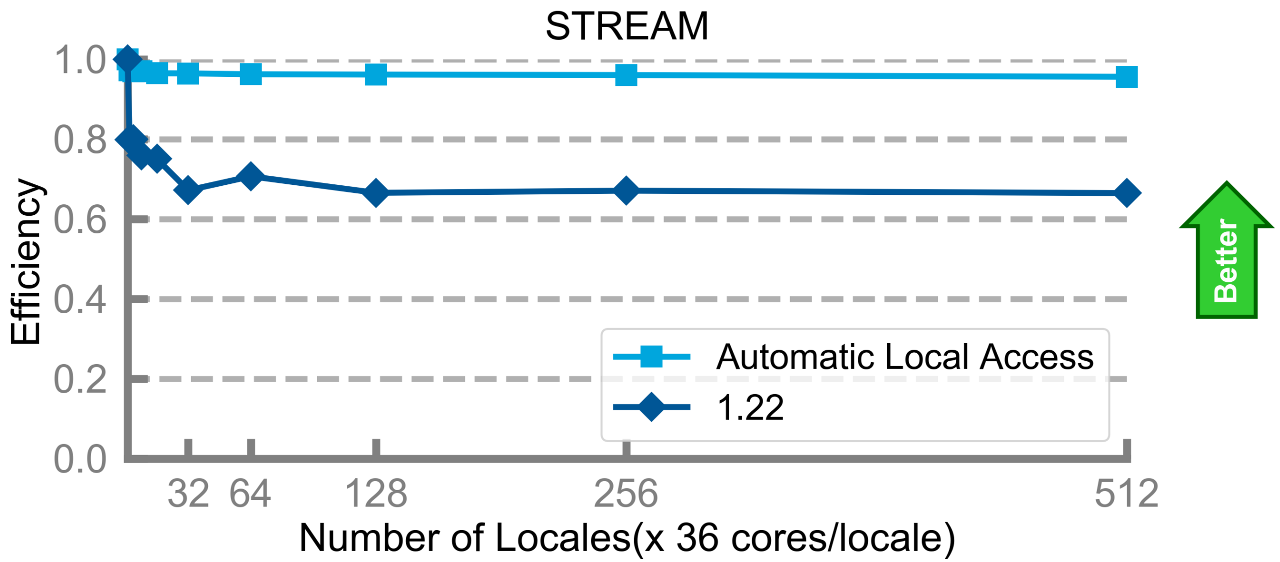
now essentially performs like other idioms:

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

or:

```
A = B + alpha * C;
```





AUTOMATIC LOCAL ACCESS OPTIMIZATION

Impact

- Explicit 'localAccess' calls are no longer needed in NPB-FT
  - Kernel with 'localAccess' calls

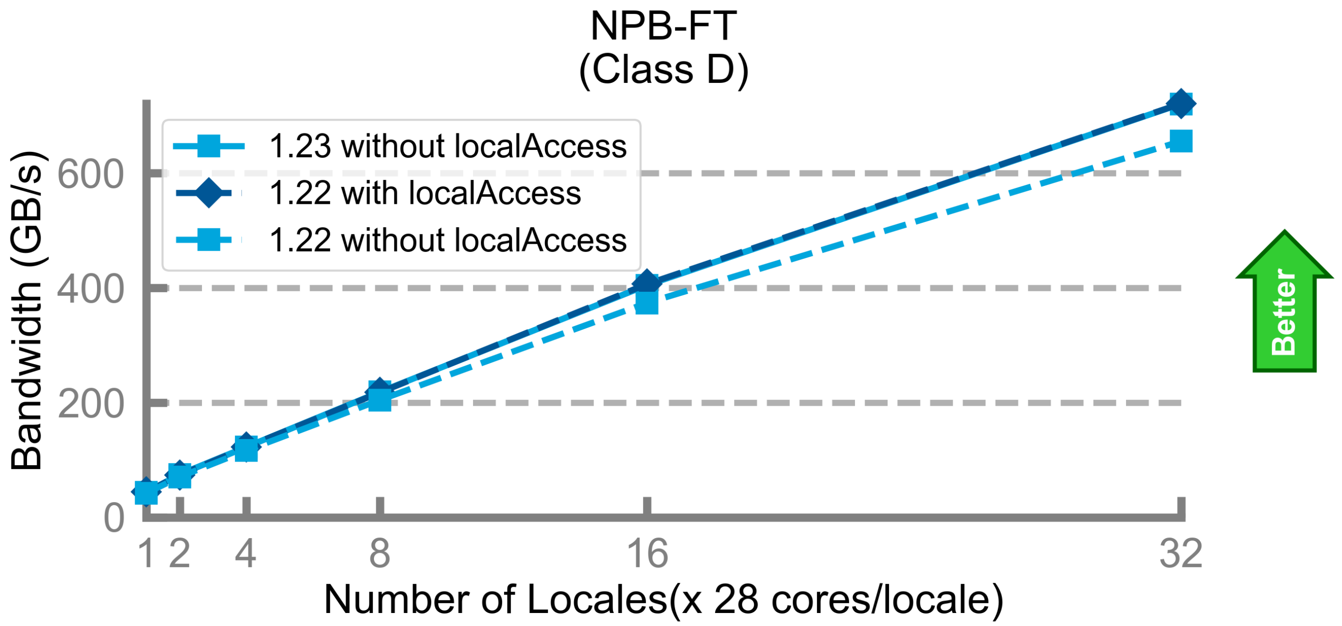
```
forall ijk in DomT {
  const elt = V.localAccess[ijk] *
              T.localAccess[ijk];

  V.localAccess[ijk] = elt;
  Wt.localAccess[ijk] = elt;
}
```

  - Kernel without 'localAccess' calls

```
forall ijk in DomT {
  const elt = V[ijk] *
              T[ijk];

  V[ijk] = elt;
  Wt[ijk] = elt;
}
```



AUTOMATIC LOCAL ACCESS OPTIMIZATION

Next Steps

- Expand static check to certain array/domain operations, e.g.:

```
coforall l in Locales do on l {
  forall i in A.localSubdomain() do    // localSubdomain always produces a subset
    A[i] = calculate(i);
  forall i in A.domain[someSlice] do  // slicing always produces a subset
    A[i] = calculate(i)
}
```

  - Accesses above will be optimized dynamically on Chapel 1.23, but we could optimize them statically

- Investigate how we can expand the analysis to affine accesses

```
forall i in A.domain do
    A[i] = calculate(A[i-1], A[i], A[i+1]);
```

IMPROVEMENTS TO ASSOCIATIVE TYPES

ASSOCIATIVE TYPES

Background and This Effort

Background: Historically, Chapel's lowest-level associative types were associative domains/arrays

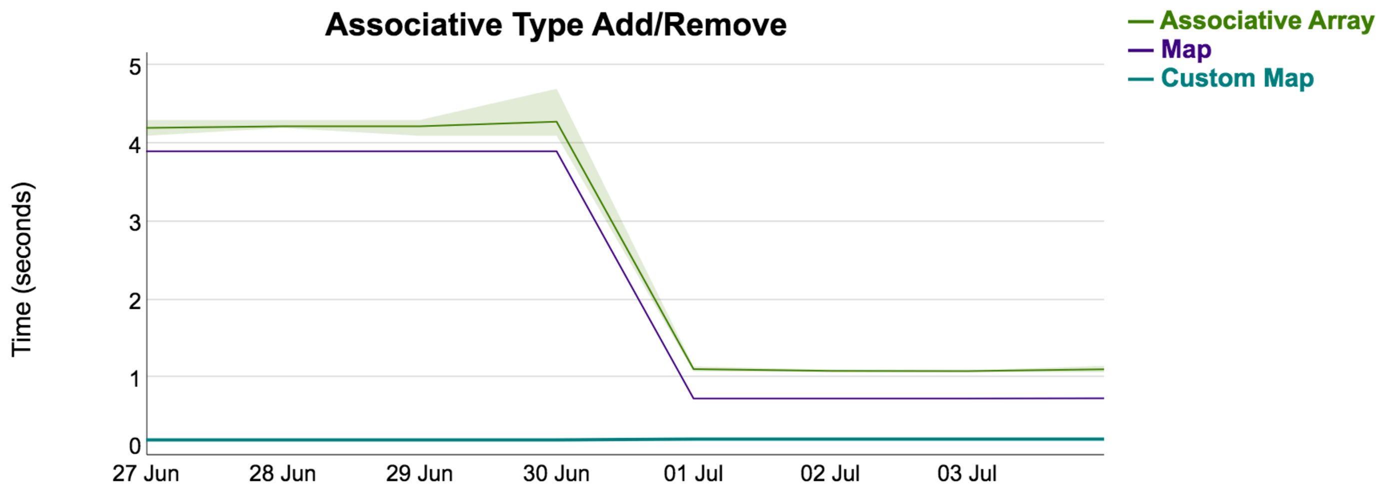
- Hash table implementation was intertwined in domain/array implementation
  - Other types like set/map were built on top of associative domains/arrays
  - Wanted associative type for internal data structures, but associative domains created circular dependency

This Effort: Factored hash table implementation into an internal standalone type

- Changed set/map types to use the standalone hash table, which enabled optimizations
- Further optimized hash table implementation, especially for repeated insertions/deletions

ASSOCIATIVE TYPES
Impact

- Significantly improved performance for associative types
  - Especially for repeated insertion/removal patterns identified by users



ARRAY TRACKING OPTIMIZATION

ARRAY TRACKING OPTIMIZATION

Background and This Effort

Background: Chapel domains track arrays declared over them

- Supports resizing arrays when their domain is modified:

```
var D = {1..10};
var A: [D] int;
var B: [D] int;
D = {1..20};         // this resizes 'A' and 'B'
```

- Previously, domains tracked arrays with a linked list, which has *O(n)* removal
- In many cases, arrays are removed in the opposite order that they are created, so *O(1)* in practice
- However, for arrays-of-arrays that freed their array elements in parallel, *O(n)* behavior occurred
  - Some user codes have suffered from this

This Effort: Switched from using a linked list to a hash table to track arrays

- Hash table insertion/removal is always *O(1)*

ARRAY TRACKING OPTIMIZATION

Impact

- Significantly reduced worst-case overheads for tracking arrays
  - ~700x speedup for task-intents with array-of-arrays

```
// Snippet from user n-body code
const nBodies = 10000;
const D = {0..#nBodies};
var forces: [D][0..#3] real;
forall d in D with (+ reduce forces) { … }          // 486.5s -> 0.65s
```

  - ~500x speedup for distributed array-of-arrays at 512 nodes

```
// Per-task timers from ISx, 9 timers in actual code
const D = newBlockDom(0..#numLocales*here.maxTaskPar);
var totalTimeSPMD, ...: [D][1..trials] real;     // 250.0s -> 0.5s
```

CONSTANT DOMAIN OPTIMIZATION

CONSTANT DOMAIN OPTIMIZATION

Background

- Tracking the arrays declared over a domain was optimized
  - However, tracking is only needed if the domain can be resized
  - Unnecessary if the domain is constant

CONSTANT DOMAIN OPTIMIZATION

This Effort

- Stop tracking arrays for domains declared 'const' or domain literals

```
const D = {1..10};
var A: [D] int;              // no need to track A, 'D' is a constant

var B: [1..20] int;      // no need to track 'B', 1..20 is a constant
```

- An important case for this optimization is array-of-arrays

```
var A: [1..1_000_000][1..5] int;     // no need to track 1 million arrays, 1..5 is a constant
```

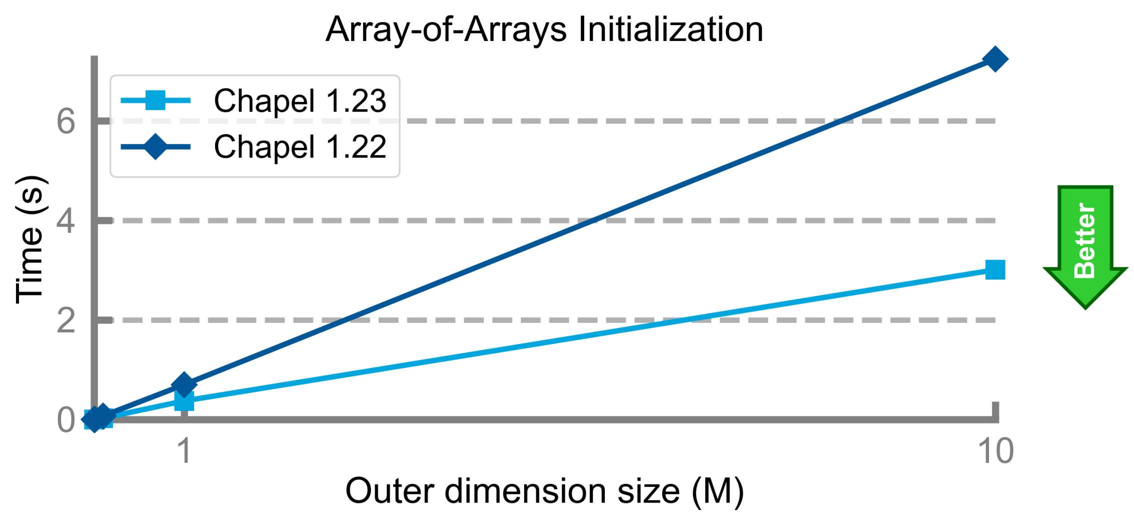
- Add compiler analysis to detect domain creation/move/copy operations
  - By *only* looking at variable/formal declarations
  - And *not* doing def/use analysis

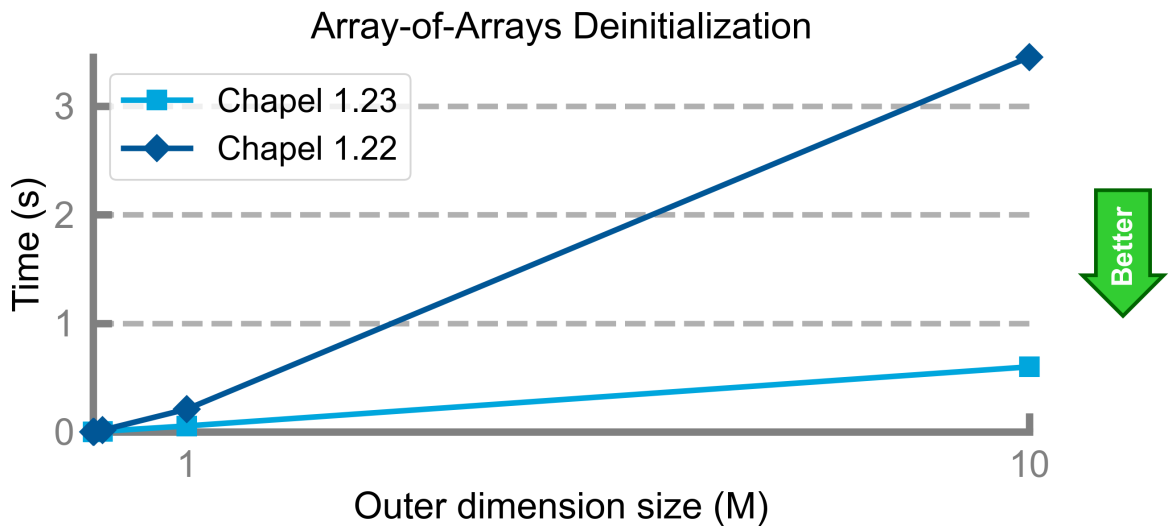
CONSTANT DOMAIN OPTIMIZATION
Impact

- More than 2x faster array initialization/deinitialization on constant domains

	Init (ns)	Deinit (ns)
Chapel 1.22	118	96
Chapel 1.23	51	47

- 2.5x faster initialization, 6x faster deinitialization for array-of-arrays





CONSTANT DOMAIN OPTIMIZATION

Next Steps

- Implement lighter-weight reference counting for domains

- More def/use analysis on domains and arrays can help cover some more cases
  - Passing a non-constant domain to a 'const ref' formal and defining an array on that formal
  - Domains that are declared 'var' but never modified

- Find answers for some semantic questions
  - Should we special-case domains w.r.t copy elision rules?
    - See https://github.com/chapel-lang/chapel/issues/16431

PARALLEL ARRAY INITIALIZATION

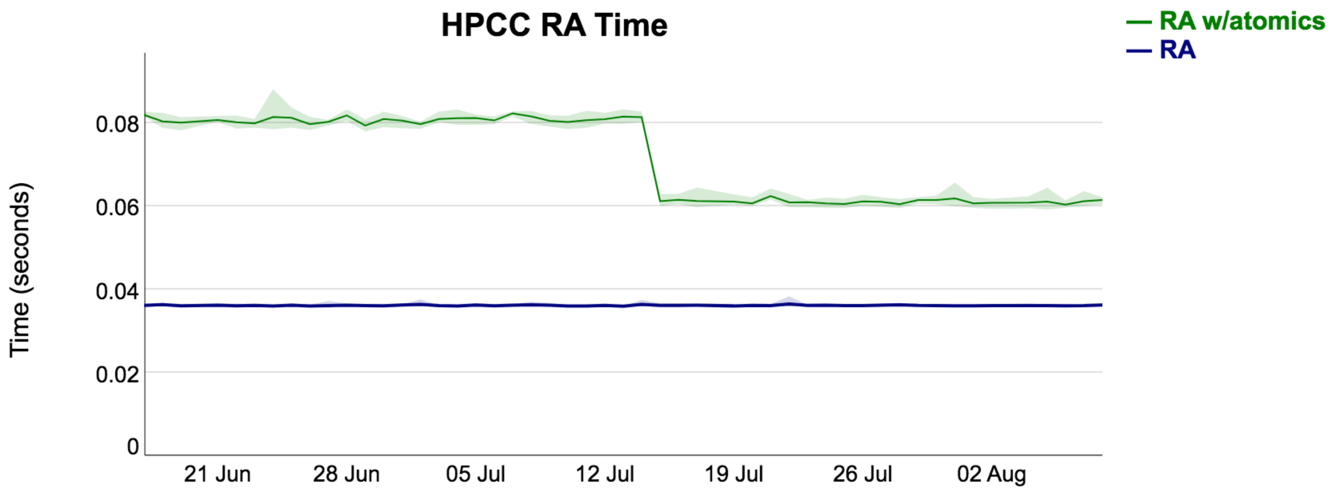
PARALLEL ARRAY INITIALIZATION

Background: Chapel initializes large numeric (integral/real/complex) arrays in parallel

- Performance issues with tracking a domain's arrays prevented parallelizing arrays-of-arrays
  - As a simplified proxy we only parallelized integral/real/complex arrays
  - Optimizing how arrays are tracked eliminated that performance issue

This Effort: Extend parallel initialization to all arrays

Impact: Better NUMA affinity for more arrays, which improves performance of parallel operations




PARALLEL ARRAY ASSIGNMENT

PARALLEL ARRAY ASSIGNMENT

Background and This Effort

Background:

- Large Chapel arrays are initialized in parallel
- However, array assignments were not parallel

```
var A: [1..n] int;    // parallel default initialization
var B: [1..n] int;    // parallel default initialization

A = B;         // this was done sequentially
```

- Especially in multi-socket systems, parallel 'memcpy's can improve the bandwidth significantly

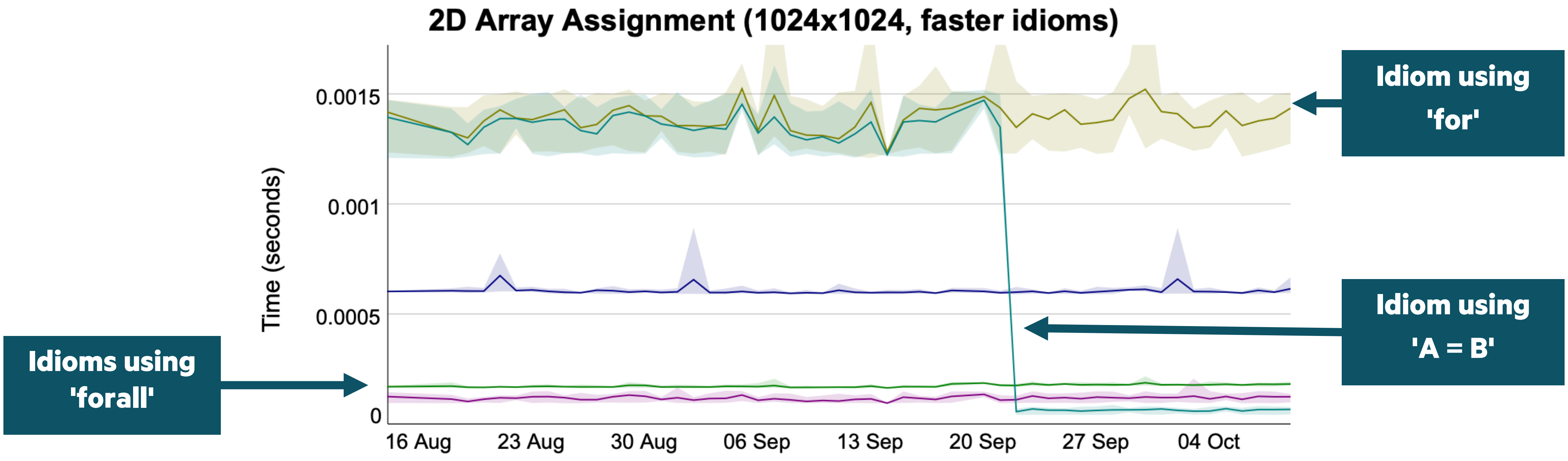
This Effort:

- Use parallel local copies for large array assignments if applicable

PARALLEL ARRAY ASSIGNMENT

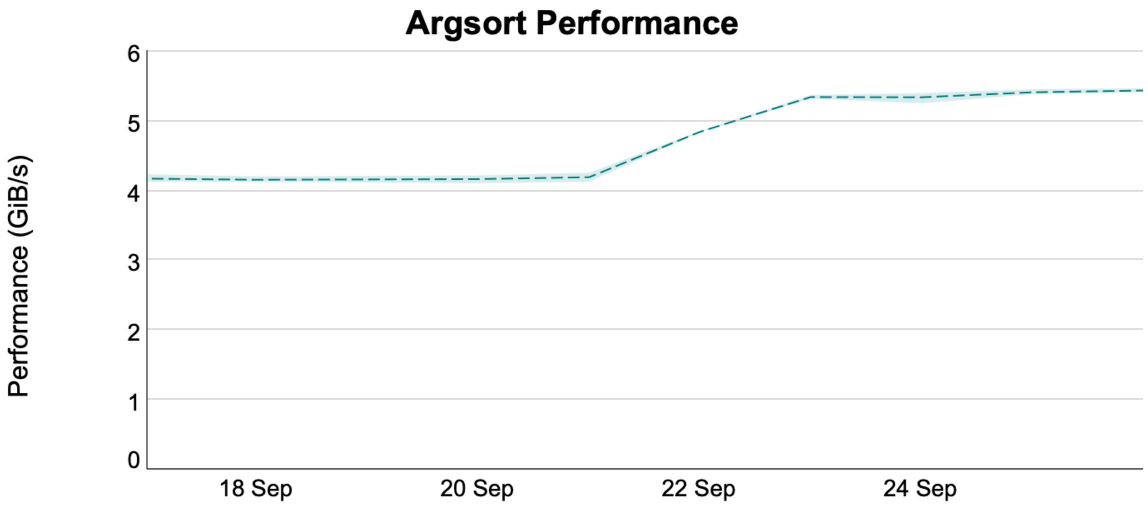
Impact

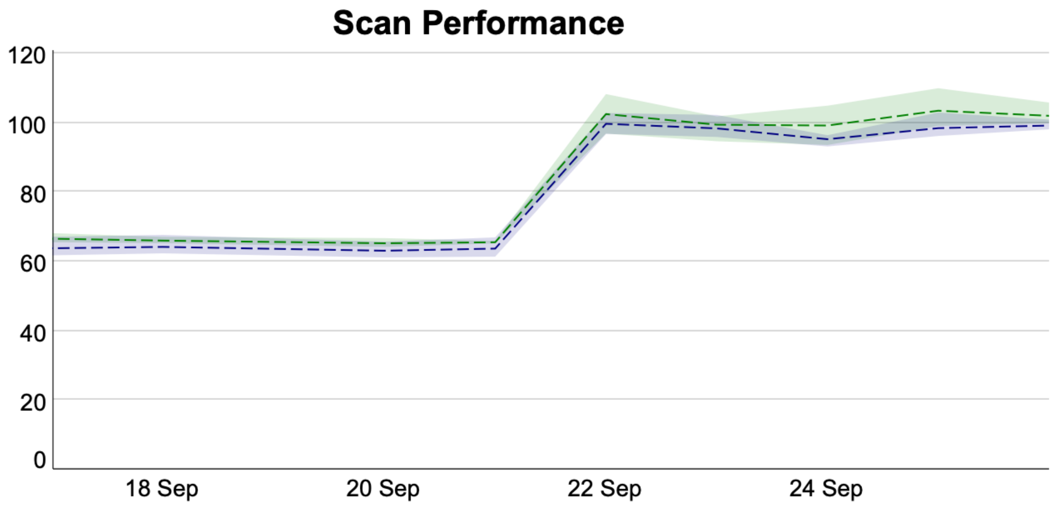
- Array copies are significantly faster



PARALLEL ARRAY ASSIGNMENT
Impact

- Arkouda performance improvements





PARALLEL ARRAY ASSIGNMENT

Next Steps

- Investigate making remote array copies parallel
  - Initial attempts resulted in some regressions

ARRAY SWAP OPTIMIZATION

ARRAY SWAP OPTIMIZATION
Background and This Effort

Background:

- Chapel supports a swap assignment operator (‘<=>’) for convenience and optimization opportunities
- Users have long requested that array swaps be performed using a pointer swap rather than per-element swaps
  - historically, this wasn’t generally possible due to our implementation of array slices
  - once we switched to using array views, it enabled this optimization in many cases

This Effort: Implemented array swaps using pointer swaps for some common cases:

- default rectangular arrays that:
  - are the same size
  - are stored on the same locale
  - are not array views
- block-distributed arrays that:
  - have equivalent distributions

ARRAY SWAP OPTIMIZATION

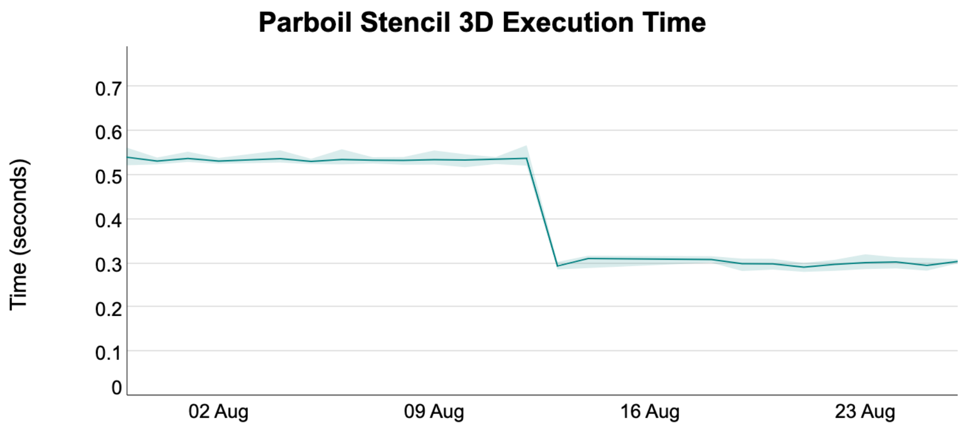
Impact

Impact: Turned array swaps for many cases from an *O(n)* operation to *O(1)* or *O(#targetLocales)*

Array size	Local Array Before	After	Factor	Block Array (16 locales) Before	After	Factor
100M	32ms	~0.15ms	213x	67ms	2.7ms	24.8x
1B	310ms	~0.15ms	2070x	510ms	3.4ms	150x
10B	[OOM]	~0.15ms	N/A	5100ms	3.2ms	1590x

- Supports writing certain code patterns more productively, such as iterative stencil patterns:

```
var New, Old: [D] real;
do {
  New = computeStencil(Old);
  const delta = max reduce abs(New - Old);
  Old <=> New;    // prepare for the next iteration
while delta > epsilon;
```



ARRAY SWAP OPTIMIZATION
Next Steps

Next Steps:

- Extend optimization to other array types and distributions
  - e.g., sparse arrays, Cyclic distributions, etc.
- Optimize other forms of array/sub-array swapping, for example:

```
A[i, ..] <=> A[j, ..];    // row swap    — think about how to implement this efficiently on distributed arrays
A[.., i] <=> A[.., j];    // column swap — (these patterns appear in PNNL's work on CHGL)
```

COMPILATION TIME IMPROVEMENTS

- Single-Iteration Coforalls
- Other Compilation Time improvements

SINGLE-ITERATION COFORALLS

SINGLE-ITERATION COFORALLS

Background and This Effort

Background: ‘coforall’ loops create a distinct task per loop iteration

- Historically, many iterators would include special cases to avoid task creation for single-iteration coforalls

```
iter batch(r: range) {
  const numTasks = here.maxTaskPar - here.runningTasks() + 1;
  if numTasks == 1 then
    for i in r do
      yield i;
  else
    coforall tid in 0..<numTasks do
      for i in myChunk(tid, numTasks, r) do
        yield i;
}
```

This Effort: Optimize single-iteration coforalls

- Avoid task creation by having parent task run body directly
- Eliminate manipulation of atomic running tasks counter

SINGLE-ITERATION COFORALLS

Impact

- Significantly faster single-iteration coforalls

```
coforall 1..1 {}                          // ~13x faster with this optimization

coforall 1..here.maxTaskPar do
  coforall 1..1 {}                        // ~90x faster with this optimization
```

- Single-iteration coforalls have little overhead now
  - Enabled removing special cases in iterators, reducing generated code size
    - ~3% faster compilation on average
    - ~15% faster Arkouda compilation

OTHER COMPILATION TIME IMPROVEMENTS

COMPILATION TIME IMPROVEMENTS

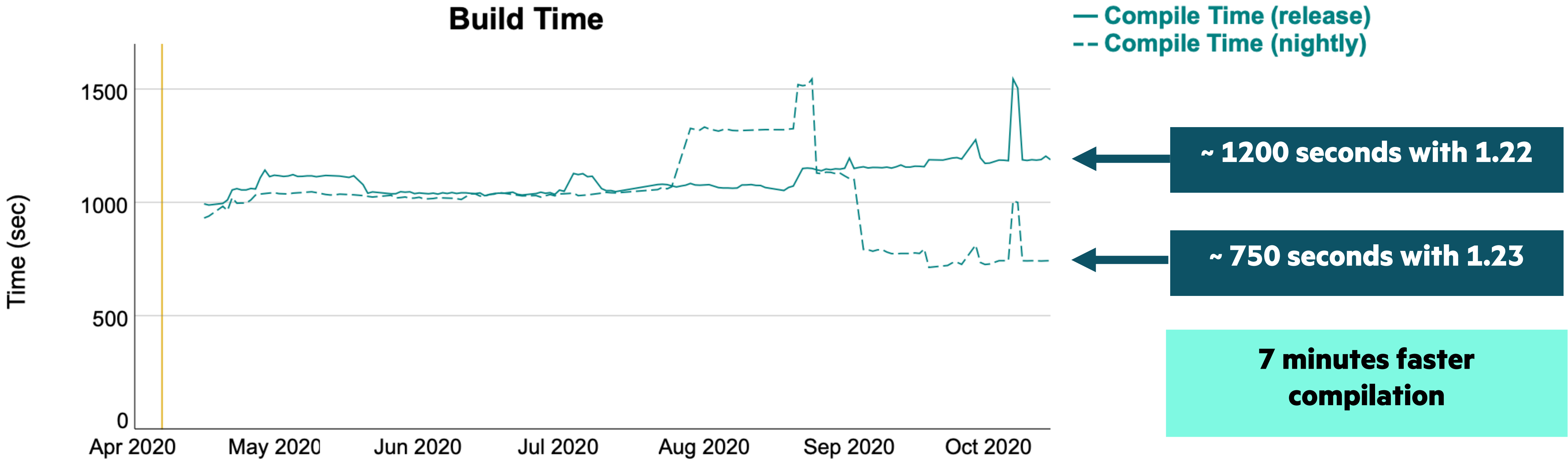
- Refactored formatted string implementation
  - Faster compilation for applications with lots of 'writef' and/or 'string.format' calls
  - ~30% faster Arkouda compilation

- Refactored several string/bytes operations
  - Reduced inlining with iterators and casts
  - ~9% faster compilation on average
  - ~3% faster Arkouda compilation

- Replaced some ‘where’-clauses with formal types
  - Fewer generic functions to resolve
  - ~7% faster compilation on average

COMPILATION TIME IMPROVEMENTS

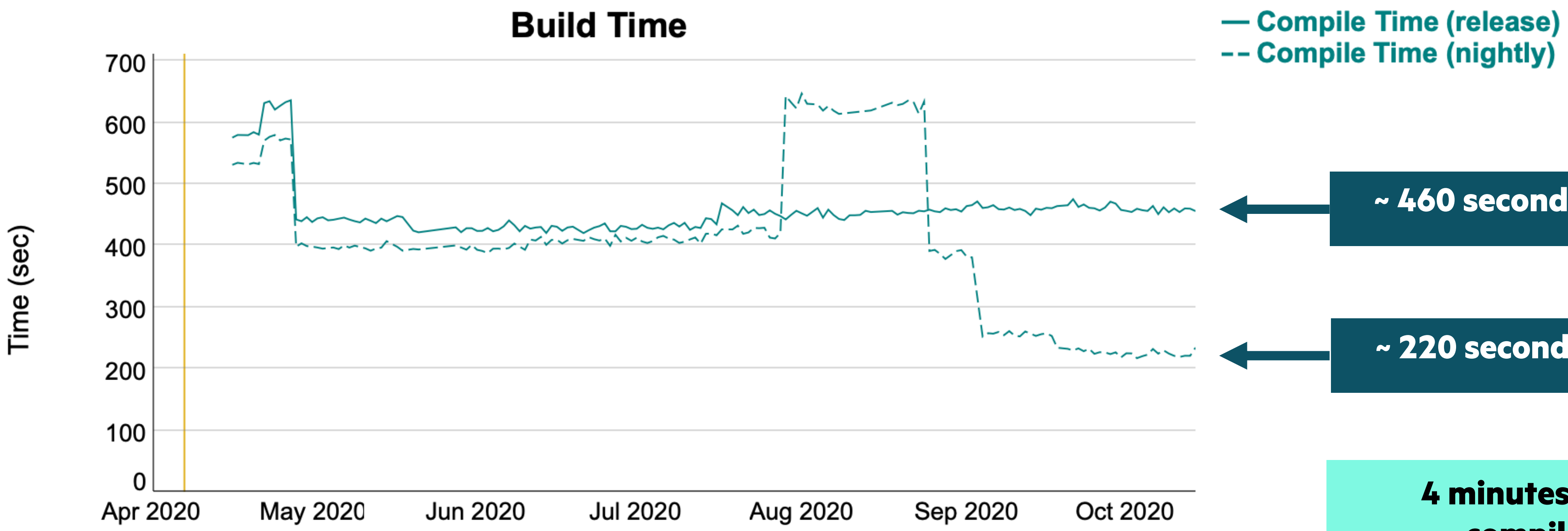
- Multi-locale Arkouda build time on Cray XC



7 minutes faster compilation

COMPILATION TIME IMPROVEMENTS

- Single locale Arkouda build time



~ 460 seconds with 1.22

~ 220 seconds with 1.23

4 minutes shorter compilation

COMPILATION TIME IMPROVEMENTS

Next Steps

- More opportunities to reduce the generated code size and compilation time
  - We can stop inlining several array support functions
    - Need to investigate potential performance regressions
  - Iterator outlining
    - There are some large iterators that we inline even with '—no-fast'
    - Currently, non-inlined iterators generate even more code and are very slow
    - Investigate whether we can outline such iterators’ bodies into helpers and inline smaller bodies

MEMORY IMPROVEMENTS

- Memory Fragmentation Improvements
- Memory Leak Improvements

MEMORY FRAGMENTATION IMPROVEMENTS

MEMORY FRAGMENTATION
Background and This Effort

Background: ‘jemalloc’ per-thread arenas can cause memory fragmentation

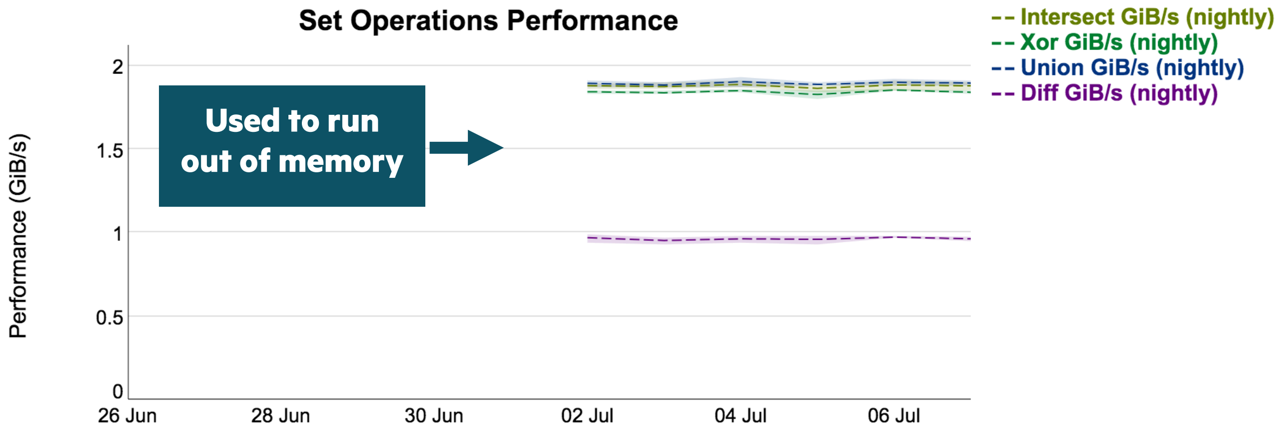
- Each thread allocates from a different arena to improve concurrent allocation performance
- Freed memory is not immediately returned to the system, but retained for later use to reduce system calls
- This leads to cross-thread fragmentation, which limits available memory for large allocations—for example:
  - thread/arena 0 allocates/frees a large array – had to grab memory from system, retains for future use
  - thread/arena 1 then does the same operation – cannot use arena 0 memory, must grab more from system
- This impacted configurations that allocate large arrays through ‘jemalloc’
  - Did not impact ugni, which uses a different allocation scheme for large arrays

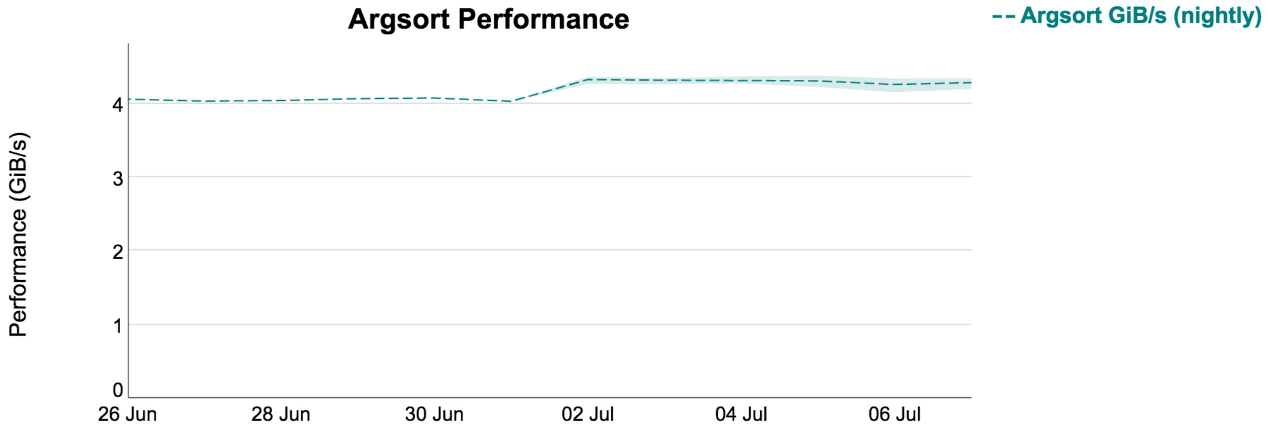
This Effort: Use a single arena to satisfy large allocations

- Increases contention for large allocations, but concurrent large allocations are rare

MEMORY FRAGMENTATION
Impact

- Reduced memory fragmentation and improved performance for repeated array creation





MEMORY LEAK IMPROVEMENTS

MEMORY LEAKS
Background, This Effort and Next Steps

Background:

- Memory leaks have historically been tracked in graphs
  - Made sense when hundreds of tests leaked
  - Makes it cumbersome to triage leaks now that there are only a few leaking tests

This Effort:

- Converted multi-locale leak testing to a correctness test now that it has 0 leaks
- Classified remaining single-locale leaks into distinct bugs with smaller reproducers
  - We believe 24 leaking tests are coming from 8 different bugs
  - See https://github.com/chapel-lang/chapel/issues/15623

Next Steps:

- Investigate turning single-locale testing into correctness tests
  - Will require some adjustments for current known/expected leaks
- Close remaining single-locale leaks

OTHER PERFORMANCE IMPROVEMENTS

OTHER PERFORMANCE IMPROVEMENTS

For a more complete list of performance optimizations in the 1.23 release, refer to the following sections in the CHANGES.md file:

- ‘Performance Optimizations’
- ‘Memory Improvements’

THANK YOU

https://chapel-lang.org
@ChapelLanguage