Hewlett Packard Enterprise

CHAPEL 1.24 RELEASE NOTES: PERFORMANCE OPTIMIZATIONS

Chapel Team

March 18, 2021

OUTLINE

- Remote Cache Improvements
- Automatic Copy Aggregation
- Automatic Local Access Improvements
- First-class Representation for Zip Clauses
- Scan Improvements
- Array Tracking Optimization
- Memory Leak Improvements

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

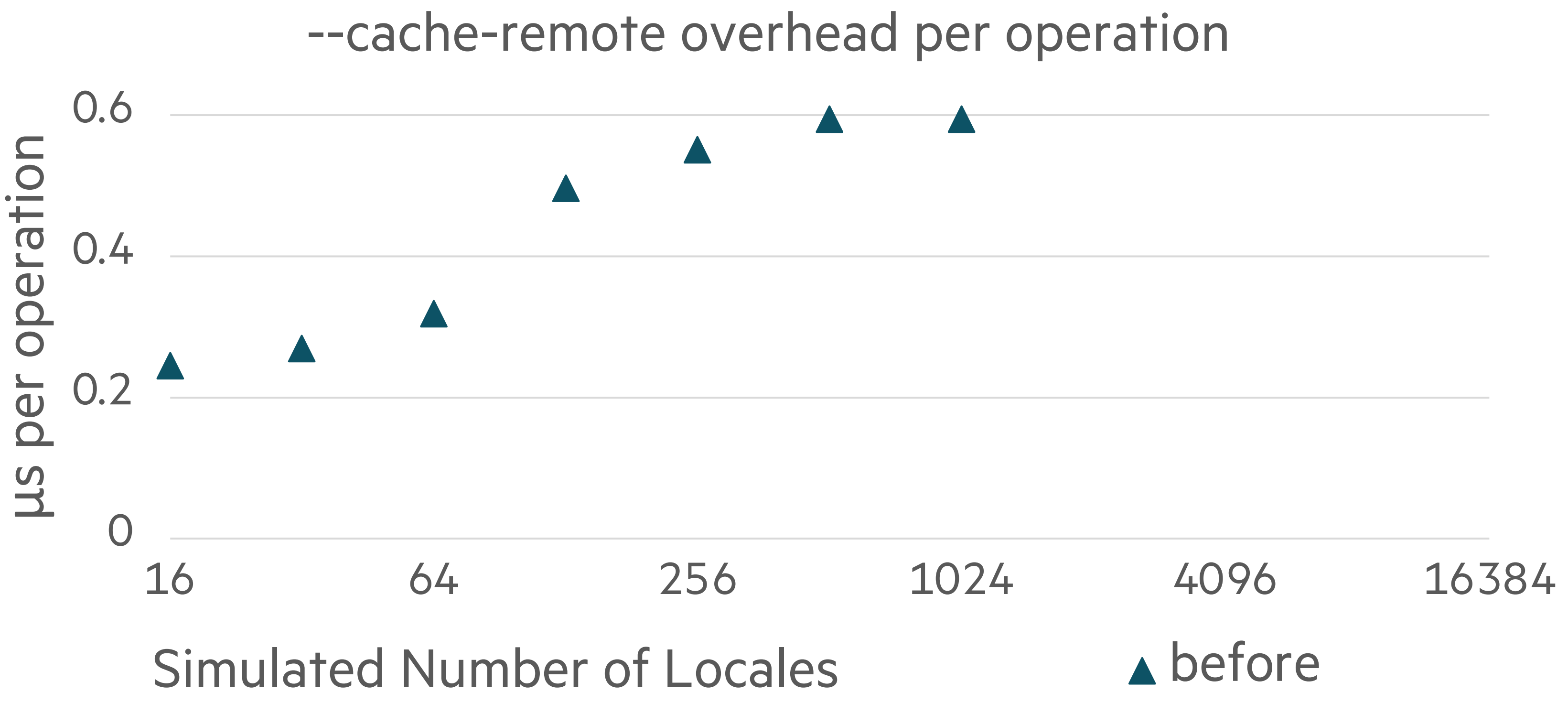
```
var A, B:[1..n] int;
on Locales[1] do
  for i in 1..n do
    B[i] = A[i];
```

	without --cache-remote	with --cache-remote
GETs for A[i]	8 bytes per iteration	1024-byte chunks
PUTs for B[i]	8 bytes per iteration	1024-byte chunks
array metadata GETs	several GETs per iteration	GETs on first iteration only

- For this toy program, cache provides 20x speedup on Cray Aries, 100x speedup on FDR InfiniBand
- Previously off by default due to performance regressions in some workloads, particularly at scale

RANDOM ACCESS CACHE SCALING
Background

- Observed scaling problems with local manipulation of cache management data structures
- Originally seen for HPCC Random Access using GETs/PUTs (RA-rmo)
- Created a synthetic benchmark that enables simulating higher locale counts to see cache overheads

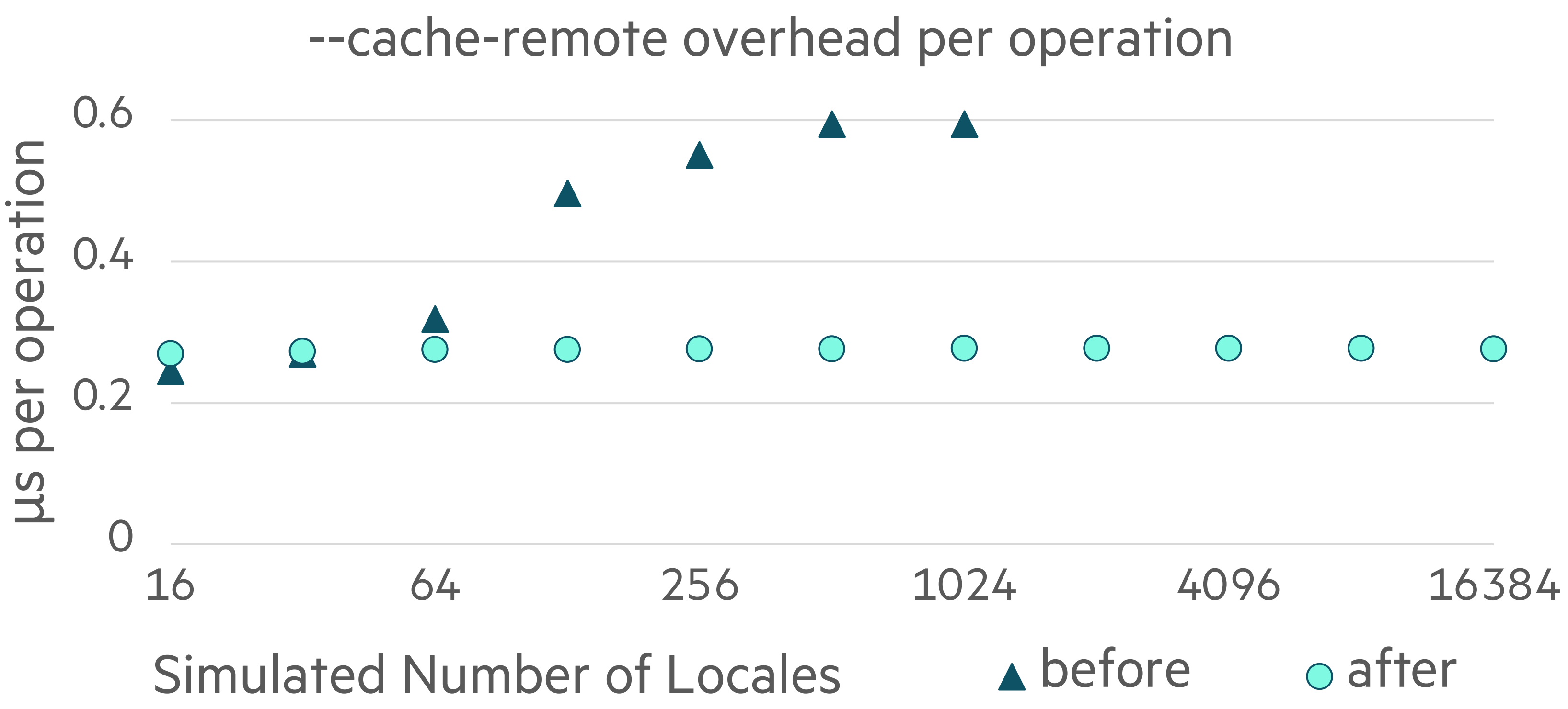


RANDOM ACCESS CACHE SCALING

This Effort and Impact

This Effort: Reduced data structure manipulation by simplifying the lookup table

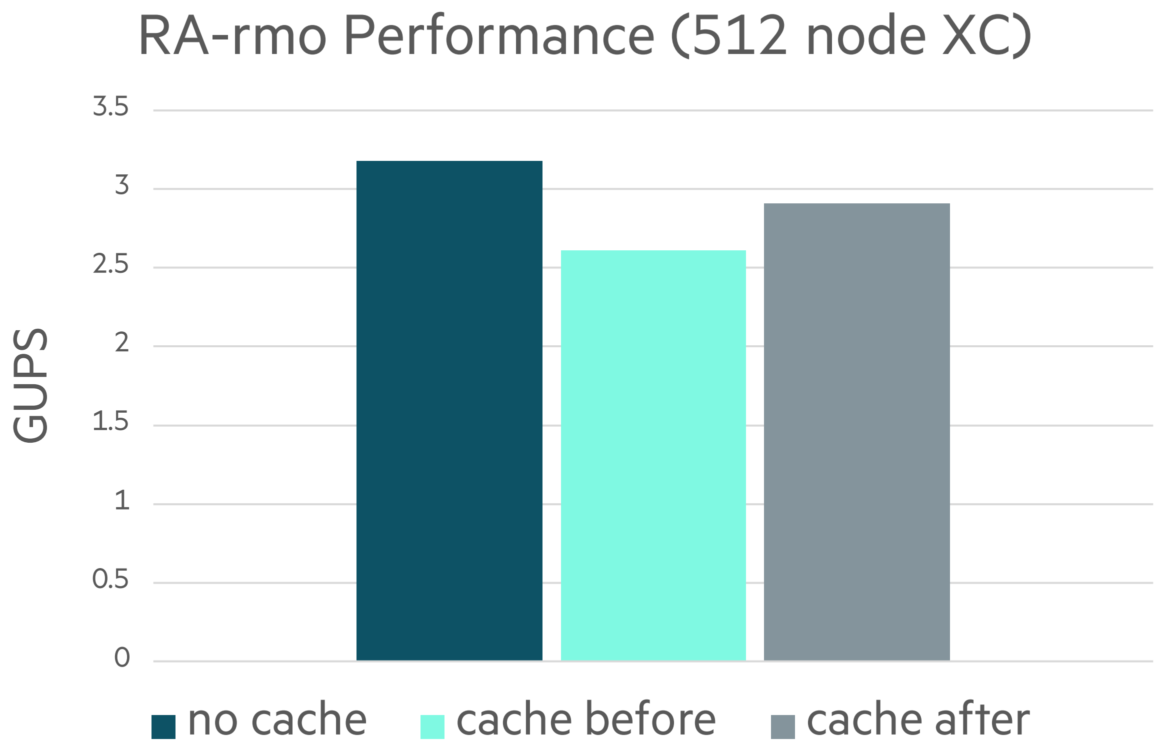
Impact: Data structure manipulation now has constant overhead



RANDOM ACCESS CACHE SCALING

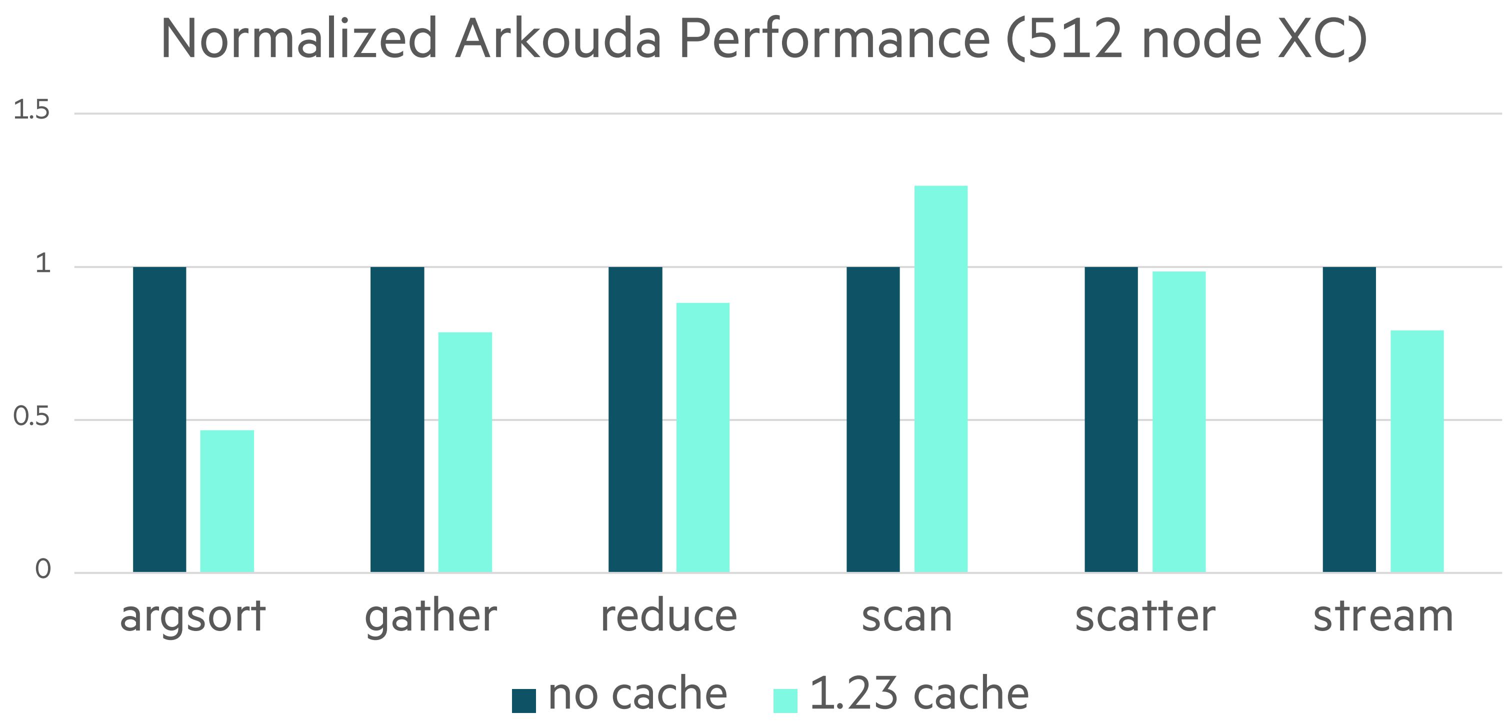
Impact

- Improved RA performance at scale
  - Still lags overall, but this is a worst-case for the cache with fine-grained random access on a fast network



ARKOUDA CACHE SCALING
Background

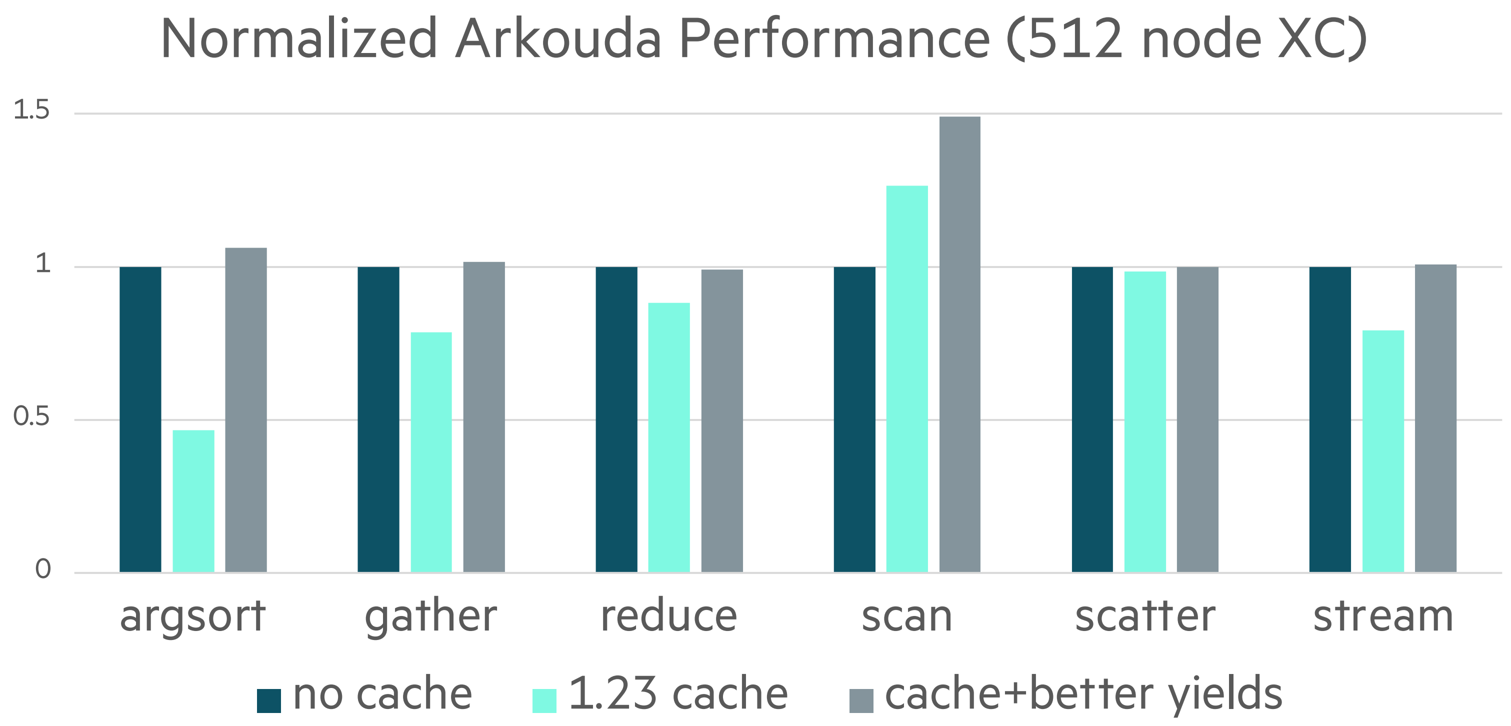
- Observed performance issues with Arkouda at scale with the cache enabled
  - Particularly for argsort and gather



ARKOUDA CACHE SCALING
This Effort and Impact

This Effort: Switched from a coarse-grained strategy to fine-grained strategy to handle task yields

Impact: Cache is now a net benefit to Arkouda performance



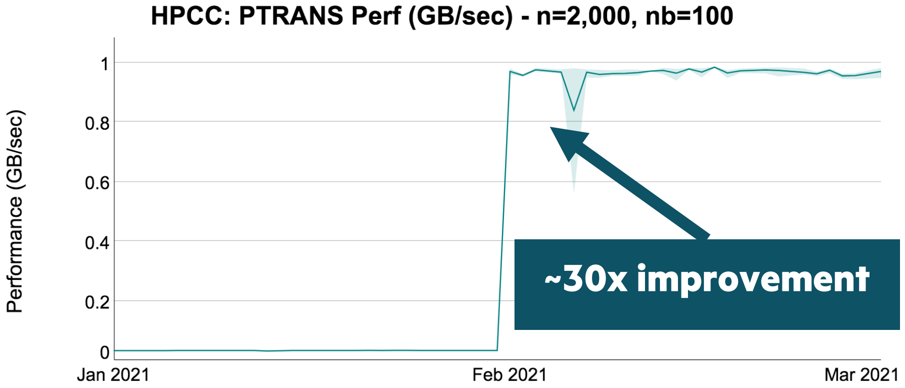
TURNING CACHE ON BY DEFAULT
This Effort

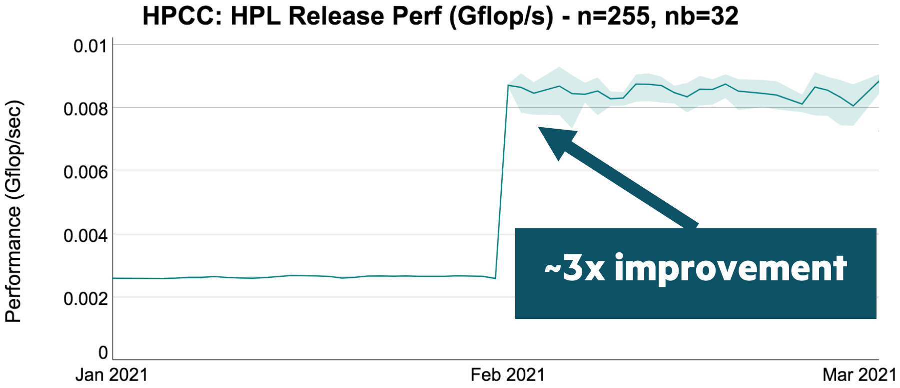
- Cache performance and scaling issues have been addressed as described in previous sections
  - Known/expected regression for RA-rmo
  - No other regressions up to 512 nodes for core benchmarks and Arkouda

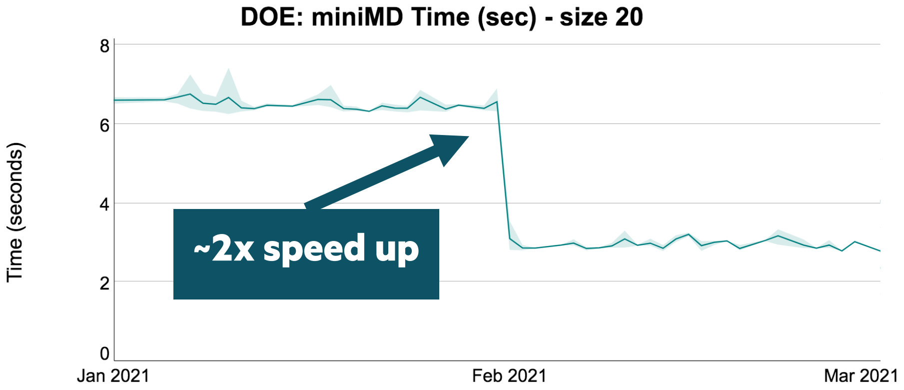
- As a result, enabled the cache by default
  - Cache can still be disabled with ‘--no-cache-remote’

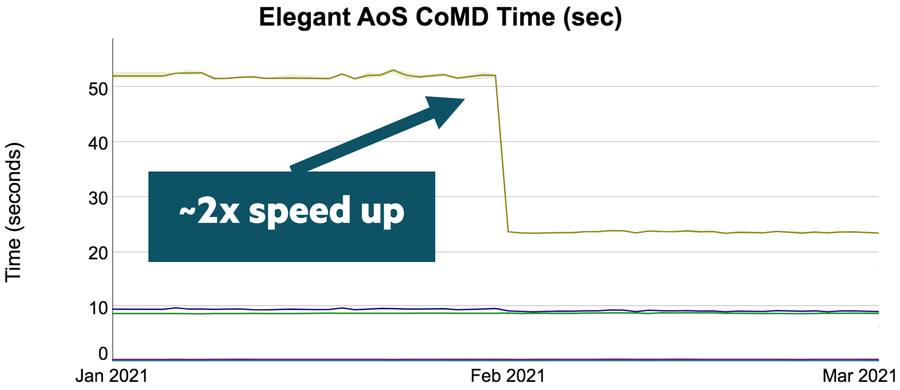
TURNING CACHE ON BY DEFAULT
Impact

- Significant performance improvements for some benchmarks



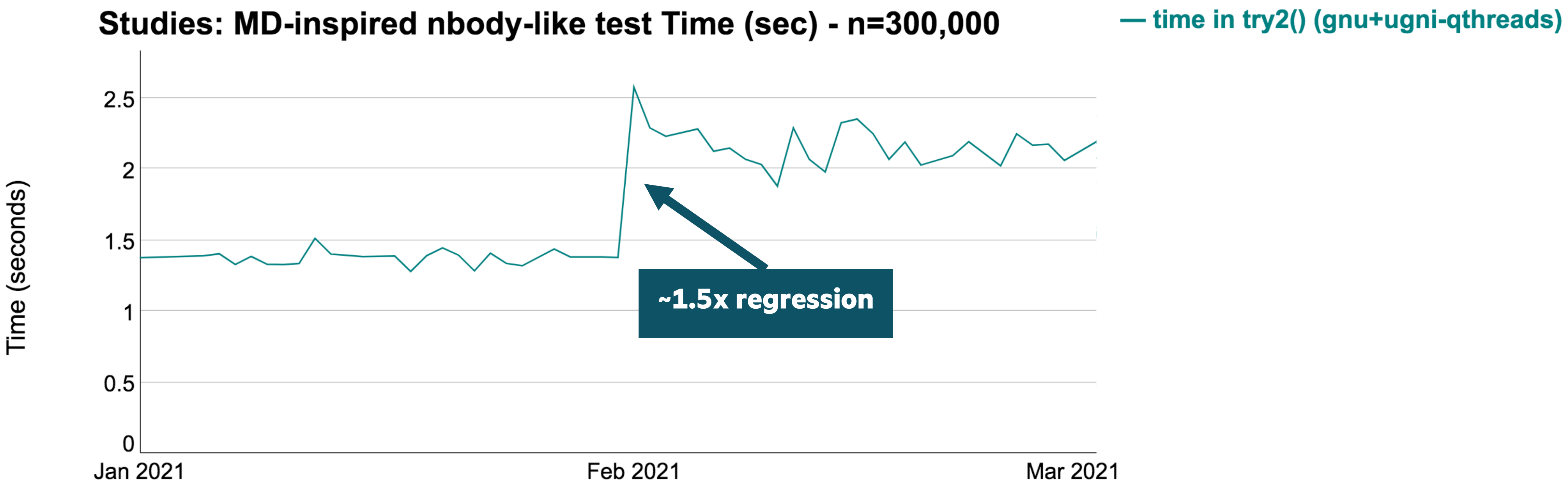






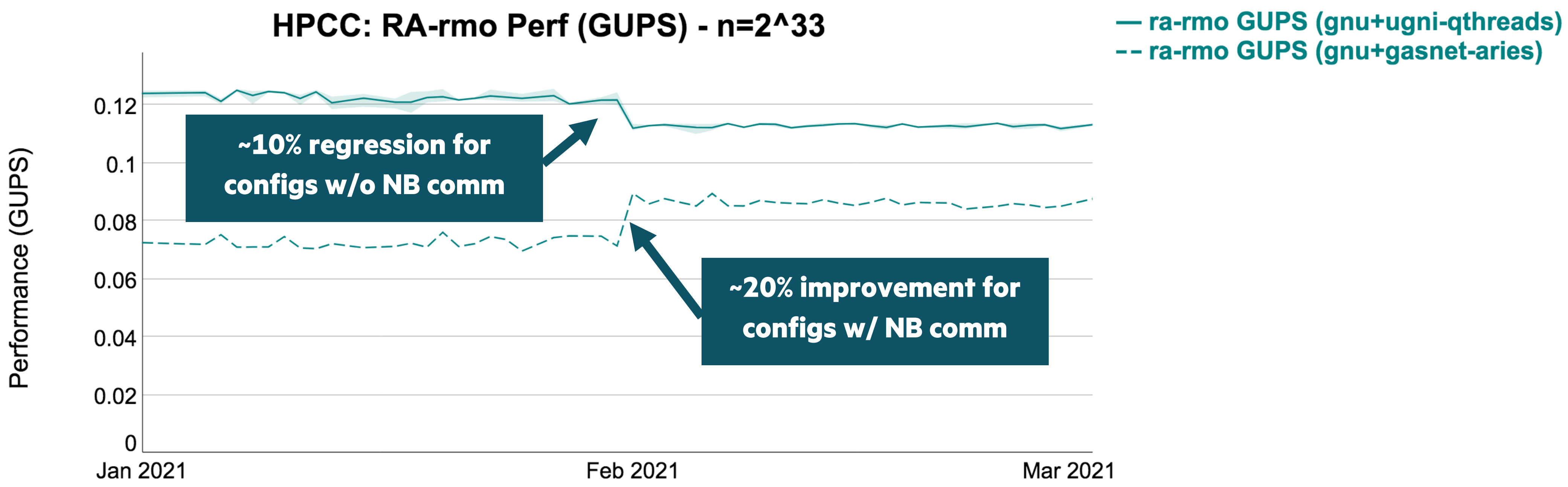
TURNING CACHE ON BY DEFAULT
Impact

- Some unexpected regressions for micro-benchmarks that have not been closely studied yet



TURNING CACHE ON BY DEFAULT
Impact

- Some expected, but minor regressions for random access patterns
  - For RA-rmo, non-blocking comm enabled by cache outweighs overhead, but only implemented for gasnet



REMOTE CACHE
Status and Next Steps

Status: Remote cache scaling has been improved and is enabled by default

- Overall, significant performance improvements for codes with suboptimal communication patterns

Next Steps: Continue to improve cache performance

- Investigate overheads for remaining performance regressions
- Implement non-blocking communication in ‘ugni’ and ‘ofi’

AUTOMATIC COPY AGGREGATION

AUTOMATIC COPY AGGREGATION
Background

- Fine-grained communication is a well-known cause of overhead in distributed memory

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;

forall i in D do
  A[i] = B[computeIndex(i)];
```

- Accesses to ‘A’ are guaranteed to be local
- Accesses to ‘B’ can be remote
  - In which case, every access will incur communication and increase overhead
  - However, these accesses can be reordered and/or aggregated

AUTOMATIC COPY AGGREGATION

Background

There are two ways this overhead can be mitigated:

1. The compiler can turn the loop into:

```
forall i in D do
  unorderedCopy(A[i], B[computeIndex(i)]);
```

   – The communication layer copies the data asynchronously, hiding the overhead
   – Can improve performance significantly
   – However, the accesses are still handled via individual messages
   – This optimization was added in 1.19, and turned on by default in 1.20

AUTOMATIC COPY AGGREGATION

Background

There are two ways this overhead can be mitigated:

2. A special “aggregator” object can be used to buffer data on its source

```
forall i in D with (var agg = newSrcAggregator(A.eltType) do
  agg.copy(A[i], B[computeIndex(i)]);
```

- Requires extra effort from the programmer
- Aggregators were first used in Arkouda during the Chapel 1.20 development cycle
  - Local data is temporarily stored in per-task, per-locale buffers
  - These buffers are of limited size and flush when they fill up
  - When they do, data is transferred in bulk
  - Achieves much better performance than unordered copy
- Arkouda source has more than 60 locations where aggregators are used

AUTOMATIC COPY AGGREGATION

This Effort

- The compiler can now use aggregators automatically with the '--auto-aggregation' flag:

```
forall i in D do
  A[i] = B[computeIndex(i)];
```

AUTOMATIC COPY AGGREGATION

This Effort

- The compiler can now use aggregators automatically with the '--auto-aggregation' flag:

```
forall i in D do
  A[i] = B[computeIndex(i)];
```

transformed into

```
forall i in D with (var agg = newSrcAggregator(A.eltType)) do
  agg.copy(A[i], B[computeIndex(i)]);
```

AUTOMATIC COPY AGGREGATION

This Effort

- This optimization is built on three existing optimizations that help in key operations and analysis

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;

forall i in D do
  A[i] = B[computeIndex(i)];
```

AUTOMATIC COPY AGGREGATION

This Effort

- This optimization is built on three existing optimizations that help in key operations and analysis

- Analysis: Can this copy be executed out-of-order?
  - Unordered forall optimization checks for hazards

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;

forall i in D do
  A[i] = B[computeIndex(i)];
```

AUTOMATIC COPY AGGREGATION

This Effort

- This optimization is built on three existing optimizations that help in key operations and analysis

- Analysis: Can this copy be executed out-of-order?
  - Unordered forall optimization checks for hazards

- Analysis: Is one side local, where the other is likely not?
  - Automatic local access does that analysis

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;

forall i in D do
  A[i] = B[computeIndex(i)];
```

AUTOMATIC COPY AGGREGATION

This Effort

- This optimization is built on three existing optimizations that help in key operations and analysis

- Analysis: Can this copy be executed out-of-order?
  - Unordered forall optimization checks for hazards

- Analysis: Is one side local, where the other is likely not?
  - Automatic local access does that analysis

- Transformation: Aggregate data
  - Arkouda has Aggregators that can be readily used

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;

forall i in D with (var agg = ...) do
  agg.copy(A[i], B[computeIndex(i)]);
```

AUTOMATIC COPY AGGREGATION

This Effort

- This optimization also supports array iterators
  - Array iterators were not subject to locality analysis in ‘forall’s
    - Automatic local access is only about ‘forall’s over domains

- Elements yielded from distributed arrays can be local within loop bodies
  - If they are part of a copy where the other side is likely remote, it is an aggregation opportunity

- The compiler can now infer that elements yielded from arrays in ‘forall’s are local within the loop body
  - This can trigger aggregation in the following case:

```
forall (a, i) in zip(A, 0..) do
  B[computeIndex(i)] = a;
```

Source of the copy is local

Copy will be aggregated

AUTOMATIC COPY AGGREGATION
This Effort

- Aggregation can also trigger based on the use of fast followers
  - When a follower is aligned with the leader, it can use the fast-follower iterator which yields elements faster
  - It also implies that yielded elements are local within the 'forall' body

```
forall (i,a) in zip(A.domain, A) do
  A[computeIndex(i)] = a;
```

'A' is aligned with the leader

'a' must be local within the body

copy can be aggregated

  - If a fast follower is used for 'A', the copy will also be aggregated
    - 'Block', 'Cyclic', and 'Stencil' distributions support fast followers
    - In many cases like the above, they will be used
    - Sometimes this can be determined statically, sometimes it relies on dynamic checks
    - In either case, if a fast follower is used, aggregation will be used as well

AUTOMATIC COPY AGGREGATION

This Effort

- --[no-]auto-aggregation
  - Enable/disable optimization
  - Off by default
  - If ‘CHPL_COMM=none’ or ‘--local’ is used, this flag is ignored, and the optimization is disabled

- --[no-]report-auto-aggregation
  - Enables/disables verbose output about the optimization steps
  - Off by default

AUTOMATIC COPY AGGREGATION

Impact

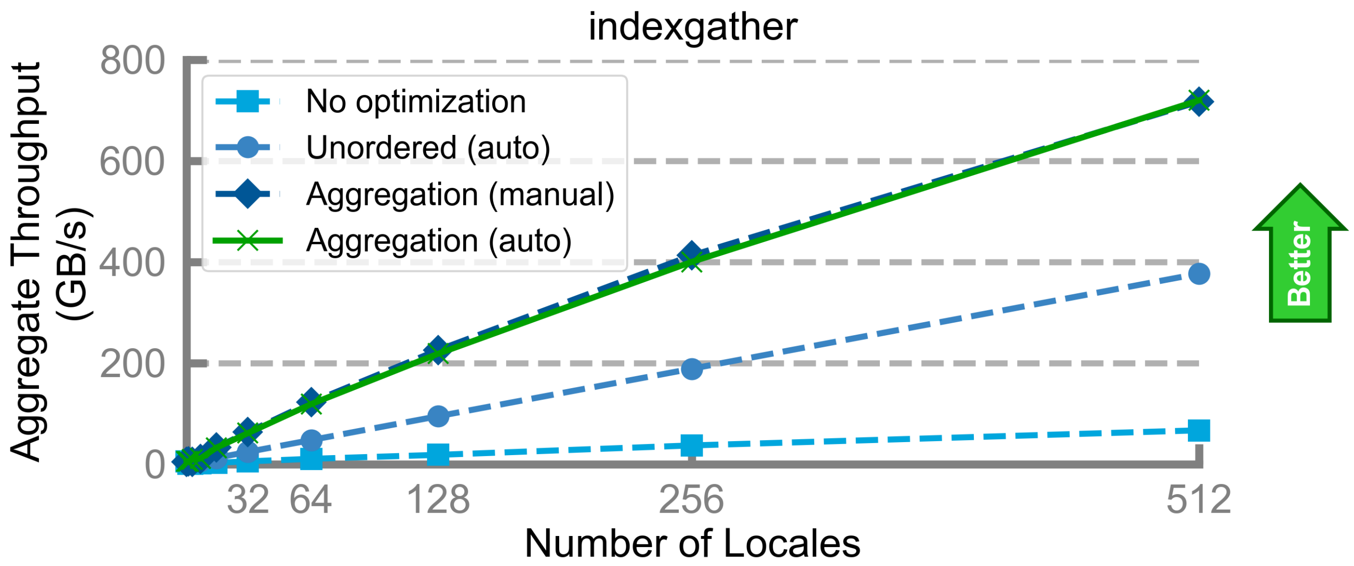
- Bale indexgather benefits greatly from aggregation
- '--auto-aggregation' reaches the same performance as the manual version
  - No user effort is needed

– Benchmark kernel:

```
forall i in D2 do
  tmp[i] = A[rindex[i]];
```

– Benchmark kernel with manual aggregation:

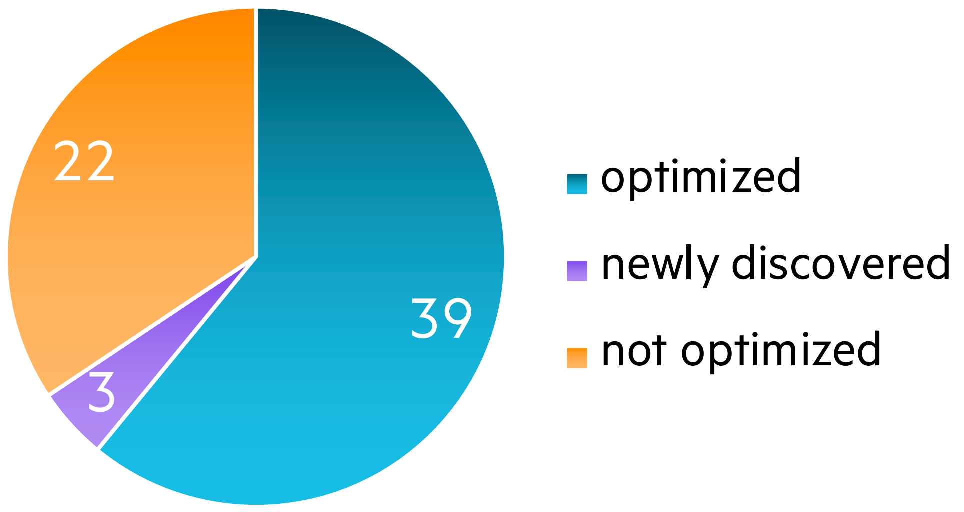
```
forall i in D with (var agg = new SrcAggregator(int)) do
  agg.copy(tmp[i], A[rindex[i]]);
```



AUTOMATIC COPY AGGREGATION
Impact

- In Arkouda, we removed all the manual aggregation from the source
  - 61 places in total
    - 39 are optimized automatically
    - 22 are not optimized
  - 3 cases that were not using aggregators are now optimized

  - The patterns where the aggregation does not fire:
    - 9: aggregation is not based on ‘forall’ loops
    - 6: compiler cannot prove that unordered operation is safe
    - 3: locality is hard to detect
    - 2: aggregated copy is not in the last statement of the body
    - 1: one side of the assignment is defined within the loop body
    - 1: needs further investigation



AUTOMATIC COPY AGGREGATION

Status

- This optimization has similar limitations to the unordered forall optimization
  - Limited to ‘forall’ bodies
    - Data parallelism implies locality in ‘forall’ bodies
    - Proving locality outside of a ‘forall’ body is more difficult
    - ‘forall’ guarantees no dependency between iterations
  - Only the last statement in the loop body is considered
    - Ensures that the destination of the copy is not used later in the body

- Fully-local aggregation causes overhead
  - Aggregators were initially implemented for manual usage in Arkouda and Bale studies
  - As such, they do not handle fully-local copies differently, causing unnecessary buffering

- Fully-remote aggregation is not supported
  - Implementing remote-to-remote aggregation is challenging

AUTOMATIC COPY AGGREGATION

Next Steps

- Reduce the overhead for fully-local aggregation
  - We can avoid buffering for cases where both source and destination are local
  - Likely to incur a dynamic branch overhead, but will avoid unnecessary buffering and copies

- Extend the optimization beyond the last statement
  - This requires detailed alias and dataflow analysis
  - However, there are some cases where a compiler-generated statement becomes the last statement
    - This unnecessarily thwarts the optimization
  - Any improvement made here will also improve the unordered forall optimization

- Reporting improvements
  - The generated report can be confusing in cases where there are multiple key statements in one line
    - Adding the ability to track the column numbers can help
    - This can also enable reproducing the source line, and marking the important parts to assist the user

AUTOMATIC LOCAL ACCESS IMPROVEMENTS

AUTOMATIC LOCAL ACCESS IMPROVEMENTS

Background

- An indexed access to a distributed array in Chapel typically works like:
  - Check if the accessed index is local
    - If so, access the local part of the array
    - Else, compute who owns that index and do a remote access
  - There are cases where arrays are accessed only locally
  - But each access incurs locality check overhead nonetheless

- In 1.23, we added a compiler optimization to automatically use ‘localAccess’
  - The optimization analyzes array accesses in ‘forall’ bodies
  - Those accesses that are aligned with the loop leader are strength-reduced
    - No need to check whether the local part of the array is accessed

```
forall i in A.domain do
  A[i] = compute(i);    // A[i] will be replaced with A.localAccess[i]
```

AUTOMATIC LOCAL ACCESS IMPROVEMENTS

This Effort

- In Chapel 1.24, accesses to array views are also covered

```
var A = newBlockArr({1..10, 1..10}, int);
ref AInner = A[2..9, 2..9];
forall i in AInner.domain do
  AInner[i] = compute(i);      // optimized in Chapel 1.24

forall i in A.domain do
  if AInner.domain.contains(i) then
    AInner[i] = compute(i);    // optimized in Chapel 1.24 (subject to dynamic checks)

ref AFirstCol = A[1..10, 1];
forall i in AFirstCol.domain do
  AFirstCol[i] = compute(i);   // optimized in Chapel 1.24
```

- We have had some implementation challenges for reindex views, so they are still not optimized

AUTOMATIC LOCAL ACCESS IMPROVEMENTS
This Effort

- Indices yielded from statically aligned followers are also considered

```
var D = newBlockDom(1..10);
var A: [D] int;
var B: [D] int;
forall (a,i) in zip(A, A.domain) do
  B[i] = compute(a);    // B[i] is replaced with B.localAccess[i]
```

- As with other cases, this optimization can be subject to dynamic checks

```
var A = newBlockArr(1..10, int);
var B = newBlockArr(1..10, int);    // cannot statically tell that `B` has the same domain as `A`
forall (a,i) in zip(A, A.domain) do
  B[i] = compute(a);    // this will also be optimized, but with a dynamic check
```

AUTOMATIC LOCAL ACCESS IMPROVEMENTS

Status

- We have a more complete automatic local access optimization in 1.24
  - Rank-change and slice views are also analyzed for the optimization
  - We leverage the fast follower optimization to analyze accesses based on indices yielded by followers

AUTOMATIC LOCAL ACCESS IMPROVEMENTS

Next Steps

- Extend the coverage to dynamically aligned followers

```
var A = newBlockArr(1..10, int), B = newBlockArr(1..10, int);
forall (a, i) in zip(A, B.domain) do
  B[i] = compute(a);              // this is not optimized in 1.24
```

- Fix issues with reindex views

- Access to shadow variables are still not covered because their scope is the loop body
  - However, this optimization adds code outside the loop to do some static and/or dynamic checks

```
forall i in myObj.A.domain with (ref innerA = myObj.A) do
  innerA[i] = compute(i);         // this is not optimized in 1.24
```

  - Can we have an outer ‘ref’ temp for ‘myObj.A’ and run checks on it?
    - This is challenging: ’myObj.A’ can have side effects

FIRST-CLASS REPRESENTATION FOR ZIP CLAUSES

FIRST-CLASS REPRESENTATION FOR ZIP CLAUSES
Background

- A zip clause indicates iteration over multiple iterables
- The compiler used to represent and implement zip clauses with tuples
  - Because historically tuple syntax was used instead of zip clauses

```
forall (a, b) in zip(A, B) do foo();
```

⇩ In 1.23, this was translated into

```
ref ARef = A, BRef = B;
var iterTuple = (ARef, BRef);
for f in getLeader(iterTuple) do
  on ... do
    for (a,b) in getFollower(iterTuple, f) do foo();
```

  - This entangled the implementation of tuples and zip clauses

- We have found this to prevent some optimization opportunities for array slices
  - When slices were used in zip clauses, they were “hidden” from the compiler and could not be forwarded

FIRST-CLASS REPRESENTATION FOR ZIP CLAUSES

This Effort

- zip clauses in forall loops now have a direct representation in the compiler

```
forall (a, b) in zip(A, B) do foo();
```

Now, this is represented with

```
for f in getLeader(A) do
  on ... do
    for (a,b) in zip(getFollower(A, f), getFollower(B, f)) do foo();
```

'A' and 'B' are used directly inside the 'on' statement

FIRST-CLASS REPRESENTATION FOR ZIP CLAUSES

Impact

- Remote value forwarding can now trigger for zipped symbols
  - Because they are no longer “hidden” in tuples
  - Takes us one step closer to making array slices lighter-weight
    - Some issues with promotions remain to be addressed

- Compilation speed improvements
  - Multilocale Arkouda compilation is ~80 seconds faster
  - Local Arkouda compilation is ~14 seconds faster

- Most of the module code that was handling zip tuples is removed
  - It was challenging to maintain

- Debugging the compiler using AST dumps and the generated code is simpler
  - The support code for zip clauses is adjacent to their foralls

FIRST-CLASS REPRESENTATION FOR ZIP CLAUSES

Next Steps

- We still use tuples to represent:
  - zippered for loops
  - forall expressions

- We plan to remove all these cases
  - Further reduction in module and generated code complexity
  - Potential improvements in compilation time
  - Easier compiler debugging

SCAN IMPROVEMENTS

SCAN IMPROVEMENTS

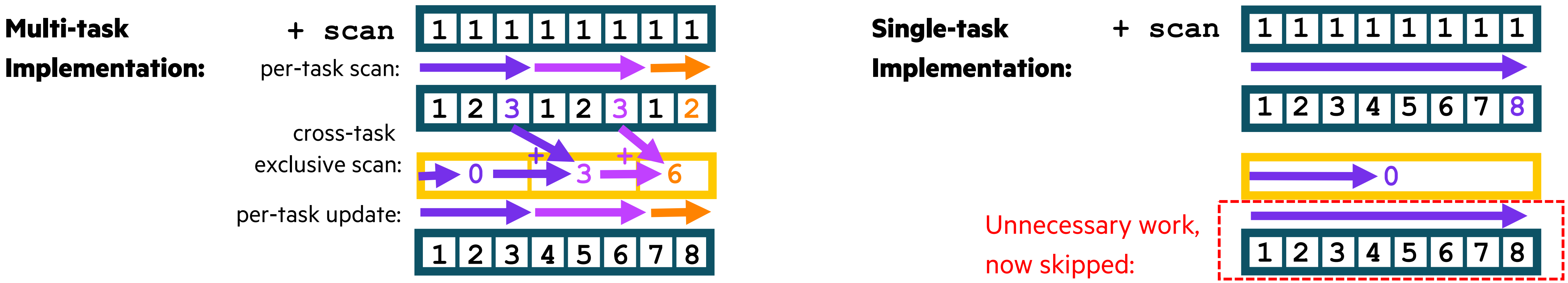
Background and This Effort

Background:

- Scans were parallelized for 1D default and Block-distributed arrays in Chapel 1.20
- Parallel scans were added for 1D Private-distributed arrays in Chapel 1.21/1.22

This Effort:

- Parallelized scan operations for 1D Replicated arrays
- Generally improved the performance of scan operations
  - squashed an unnecessary default initialization of the result array
  - skipped the second "update" pass when using a single task for scans of local arrays:

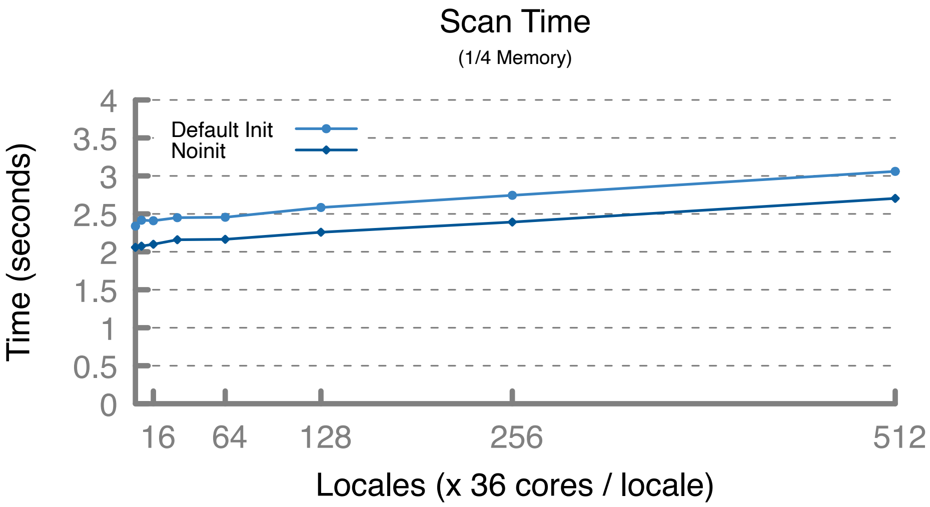


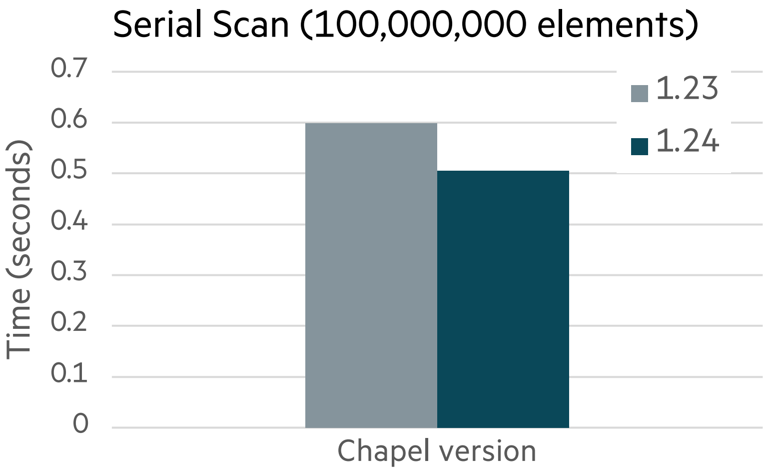
SCAN IMPROVEMENTS
Impact

Impact:

- Scans on Replicated 1D arrays should see improved performance / a lack of serialization warnings
- Most 1D scans should see modest performance benefit
  - particularly those on local arrays within parallel code sections:

```
var A, B: [1..m] [1..n] real = …;
forall i in 1..m do
  B[i] = + scan A[i];    // here, each scan will typically use a single task since the 'forall' is likely to utilize all the cores
```





SCAN IMPROVEMENTS
Status and Next Steps

Status:

- Parallel scans of 1D arrays are increasingly well-supported and tuned

Next Steps:

- Improve scalability of block-distributed scans
- Parallel scan improvements:
  - ensure scans of 1D array-like expressions are parallelized

    ```
    B = + scan (A: int);
    ```
  - parallelize scans of multidimensional arrays
  - consider extending parallelism to challenging/less mature distributions (e.g., Cyclic, Block-Cyclic)
  - generalize implementation to support cases where the ‘result’ and ‘state’ types don’t match
- Add language support for partial scans, exclusive scans, directional scans
- Finalize and document the user-defined reduction/scan interface

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'
```

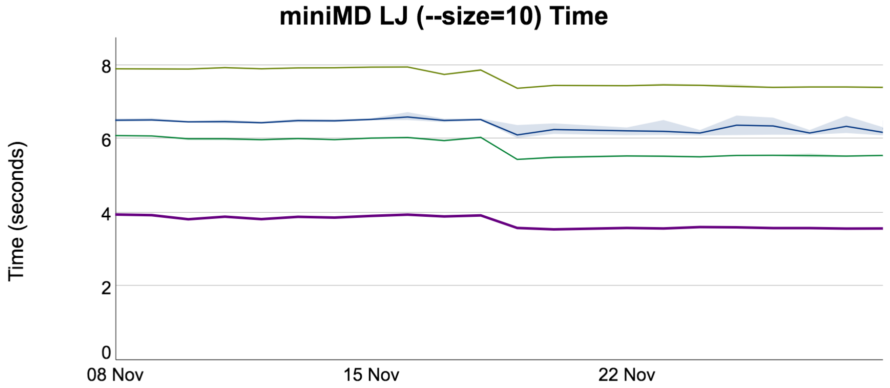
- Prior to Chapel 1.23, domains tracked arrays with a singly linked list—O(1) insertion, O(*n*) removal
- Chapel 1.23 switched to a hash table to track arrays—O(1) insertion and removal
  - Significantly reduced worst-case tracking behavior, but slightly hurt best-case and increased compilation time

This Effort: Switched from hash table to a doubly linked list

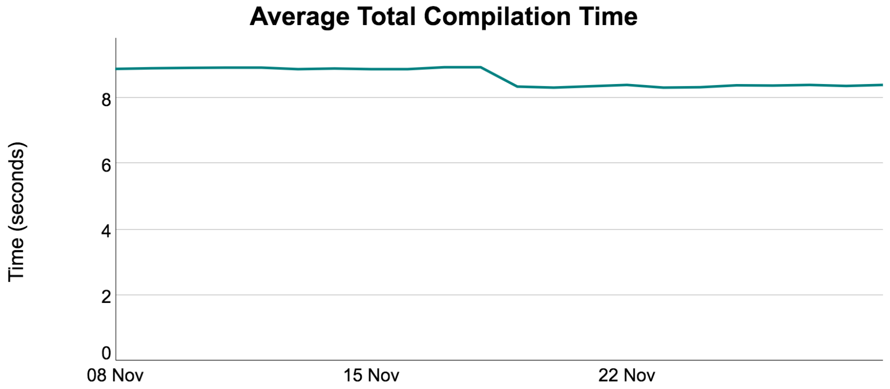
- O(1) insertion and removal still, but with a much smaller constant factor

ARRAY TRACKING OPTIMIZATION
Impact

- Faster array view creation due to reduced tracking overhead



- Reduced compilation time because hash table is no longer compiled by default



MEMORY LEAK IMPROVEMENTS

MEMORY LEAK IMPROVEMENTS

Background:

- We have been working on closing compiler-generated memory leaks
  - In 1.23, we had 24 leaking tests that were caused by 8 distinct bugs

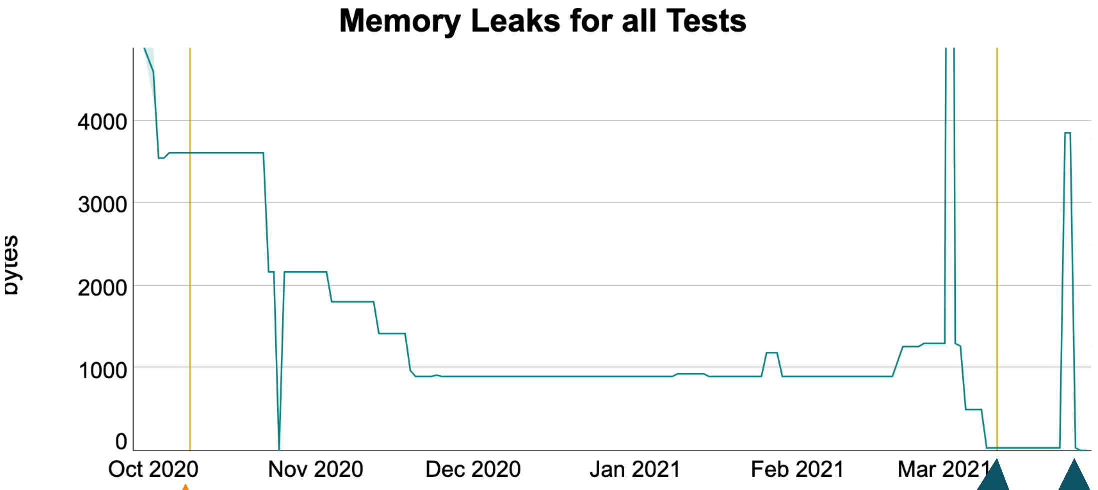
This Effort:

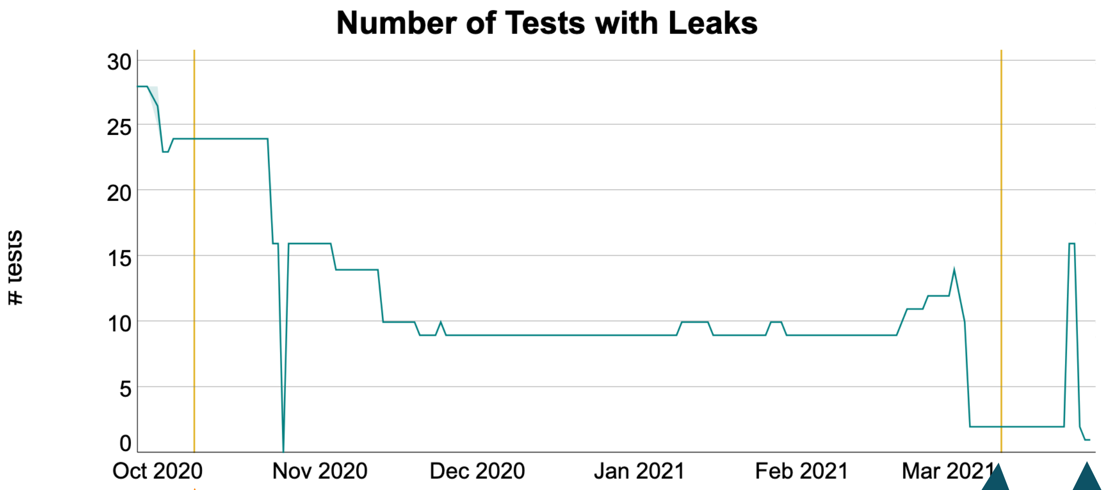
- 1.24 closes all known remaining leaks

MEMORY LEAK IMPROVEMENTS

Impact:

- As of Chapel 1.24.1, we do not have any known leaks remaining
  - A single test leaks by design, but is not currently factored out of the memory leaks test system
  - In 1.24.0, there was one last-minute leak that stemmed from the new ‘interface’ feature, but it has since been closed





3.6 KB leaked in 1.23

40 B leak in 1.24.0

8 B leak in 1.24.1

24 tests leaked in 1.23

2 tests leak in 1.24.0

1 test leaks in 1.24.1

MEMORY LEAK IMPROVEMENTS

Status:

- There are no known memory leaks remaining as of Chapel 1.24.1

Next Steps:

- Adjust the testing infrastructure to report any new leaks as a correctness failure
- Develop best practices for package authors to do memory leak testing
- Extend nightly memory leaks testing to both backends
  - Today, we test for memory leaks with the LLVM backend only manually and on occasion

OTHER PERFORMANCE IMPROVEMENTS

OTHER PERFORMANCE IMPROVEMENTS

For a more complete list of performance changes and improvements in the 1.24 release, refer to the following sections in the CHANGES.md file:

- ‘Performance optimizations/improvements’
- ‘Memory improvements’

THANK YOU

https://chapel-lang.org
@ChapelLanguage