Hewlett Packard Enterprise

Chapel 2.1 / 2.2 Release Notes: Performance Status and Optimizations

Chapel Team

June 27, 2024 / September 26, 2024

Outline

- Activities Overview
- Scalability Since 1.32
- Regressions and Resolutions
- New Optimizations

Activities Overview

Overview of Activities for 2.1 and 2.2

Optimizations:

- Const domain localization
- Optimizing away array allocations for moves
- Array view elision
- Stencil distribution improvements

Nightly Testing:

- Added nightly perf testing for HPE Cray EX platform
- Added nightly co-locale perf testing
- Added public links to nightly Arkouda test results
  - serial (Cray CS) and parallel (Cray XC)

Performance Regressions and Resolutions

- ISx HPE Cray EX hang introduced in 2.1; fixed in 2.2
- ra-rmo HPE Cray EX regression in 2.1; partial fix in 2.2
- ra-on HPE Cray EX regression in 2.1; to be fixed in 2.3

Other Activities:

- Scalability studies of core benchmarks on HPE Cray EX (SS11), Apollo (IB), and CrayCS (Aries) platforms

Outreach:

- Blog posts on
  - Navier-Stokes
  - Billion row challenge
- Call for virtual pair-programming sessions w/ Python Programmers who wish to improve speed/scalability
- Publications authored by Chapel users:
  - Josh Milthorpe et al. “Performance Portability of the Chapel Language on Heterogeneous Architectures" Heterogeneity in Computing Workshop (HCW)
  - Tiago Carneiro et al. “Investigating Portability in Chapel for Tree-Based Optimizations on GPU-powered Clusters” Europar 2024

Scalability Since 1.32

Scalability Since 1.32
Background

- In 1.32 release notes we presented scalability results of “core” benchmarks in Chapl
  - In our 1.33 and 2.0 release notes we did not present on the scalability of these benchmarks
  - We want to ensure that since then our performance has been maintained or improved
  - In these slides we show recent performance and compare it to our last reported historical performance (1.32)
- The systems we used to gather our data are:
  - HPE Cray EX / SS11 hardware
    - Dual-socket Milan (128 cores total)
    - Single 200 Gbps NIC
  - HPE Apollo / InfiniBand hardware
    - Dual-socket Xeon 8360Y (72 cores total)
    - Single 200 Gbps NIC
- In 1.32 release notes we also looked at historical Cray XC (Aries) hardware.
  - Internally, we gathered Cray XC scalability results that showed maintaining performance
  - We exclude those results from this deck since the hardware is older

Scalability Since 1.32
Background

- For comparison, we regathered 1.32 results because:
  - We no longer have access to the InfiniBand-based machine we used for 1.32
  - There have been various system software updates
  - Many of the 1.32 release-note graphs were generated to study various configurations options
    - which is tangential to the “have things regressed” comparison we’re aiming to answer with these slides

- Benchmarks we look at are:
  - **Stream**: No communication aside from task startup/teardown, NUMA affinity sensitive
  - **ISx**: Concurrent bulk communication over wide address range, NUMA affinity sensitive
  - **Bale Indexgather**: Concurrent get-style communication
    - for this benchmark we look at “fine grain” performance and performance when using Chapel aggregators
  - **RA**: Concurrent random fine-grained updates over wide address range
    - we have three different versions: get/put vs. active message vs. remote atomics
  - **Arkouda Argsort:** aggregated movement of array indices in support of sorting

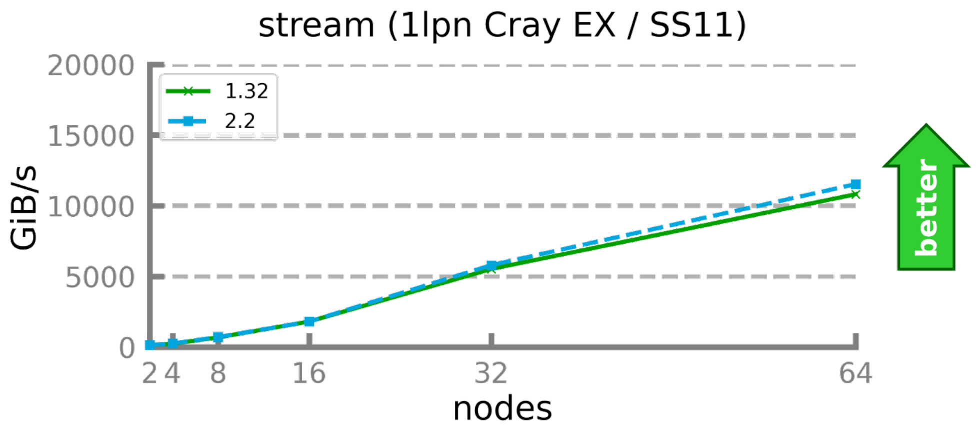
Scalability Since 1.32

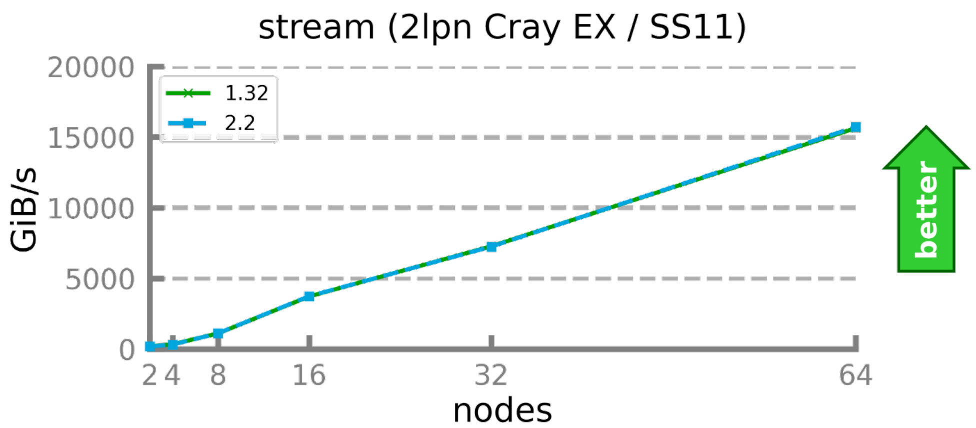
Stream Performance

- Linear scaling across nodes; similar scaling across Chapel versions

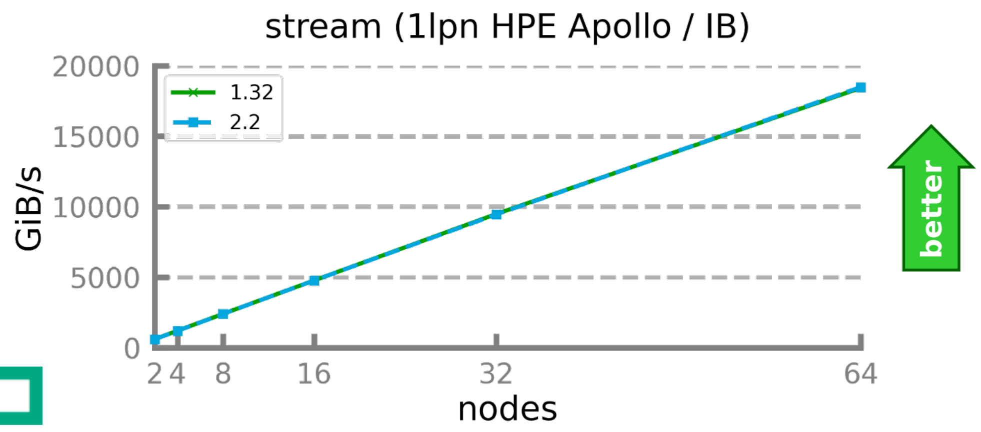
1 locale-per-node | **2 locales-per-node**

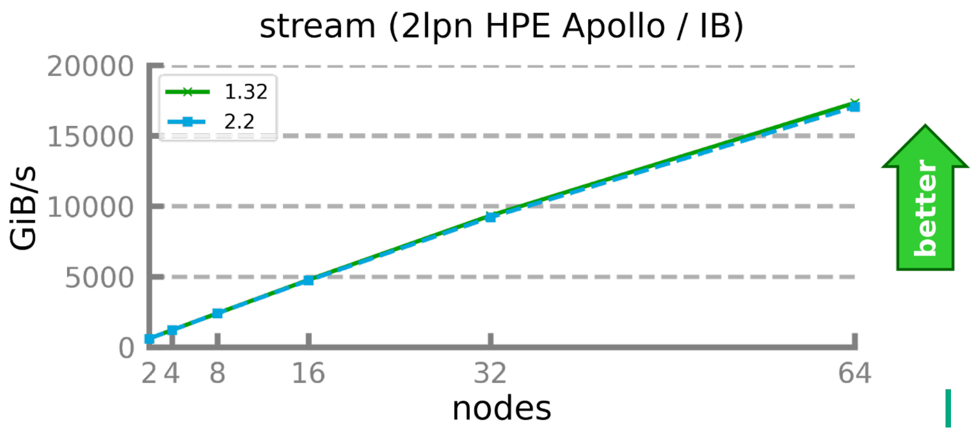
EX/SS11





Apollo/IB



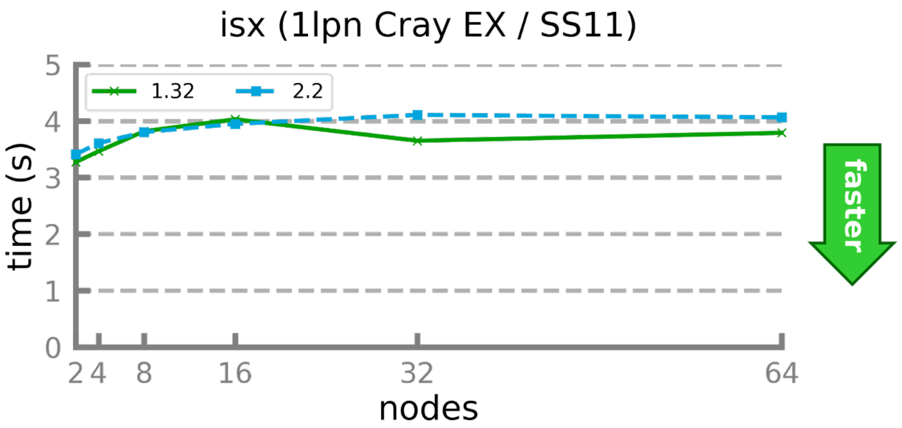


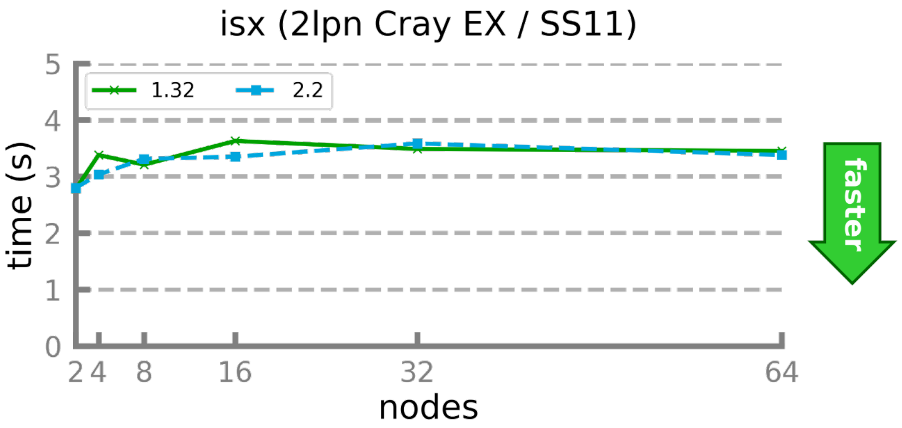
Scalability Since 1.32
ISx Performance

- The benchmark uses weak scaling (so flat profile is desired); overall profile similar across releases

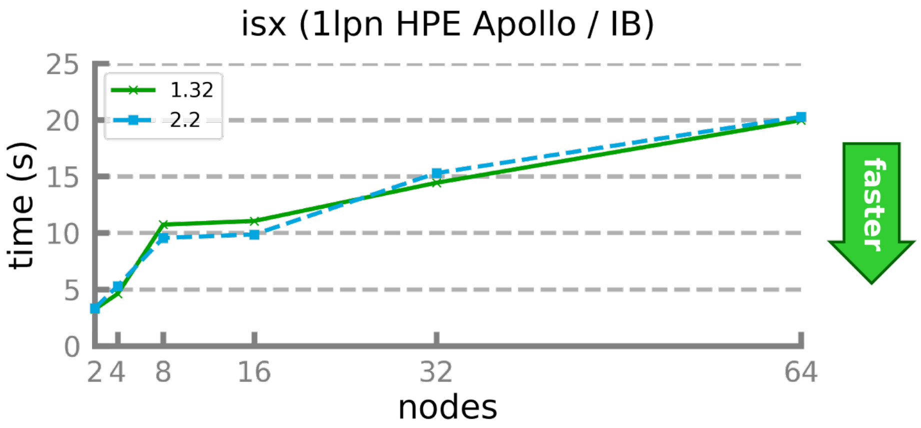
1 locale-per-node | **2 locales-per-node**

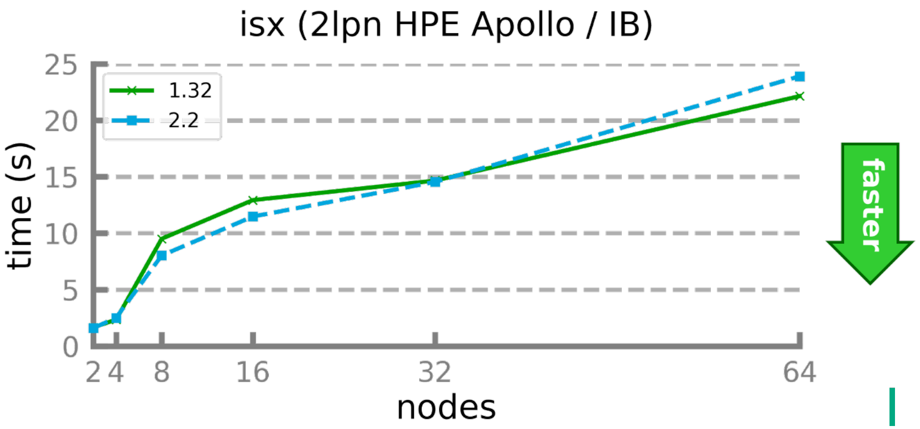
EX/SS11





Apollo/IB





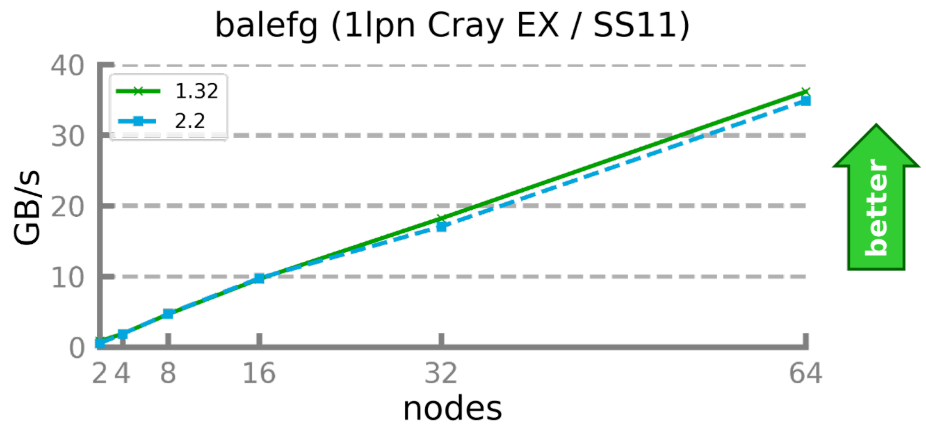
Scalability Since 1.32

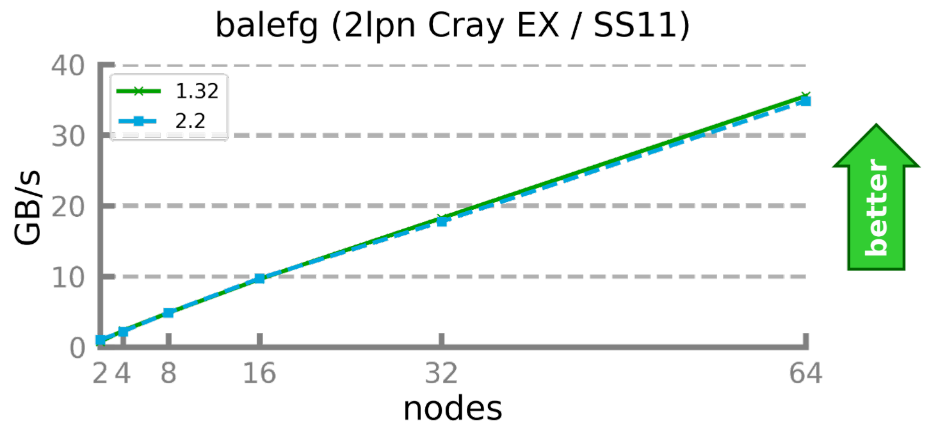
Bale IndexGather – Fine-Grained Performance

- Linear scaling across nodes; similar scaling across Chapel versions

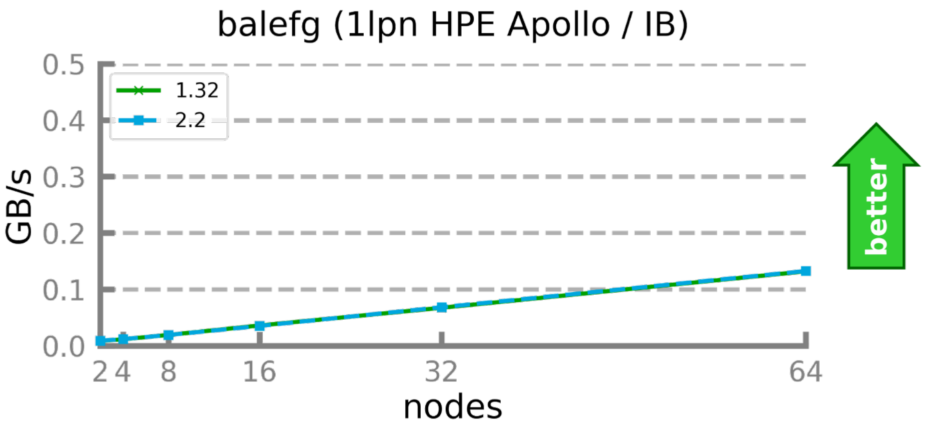
1 locale-per-node | **2 locales-per-node**

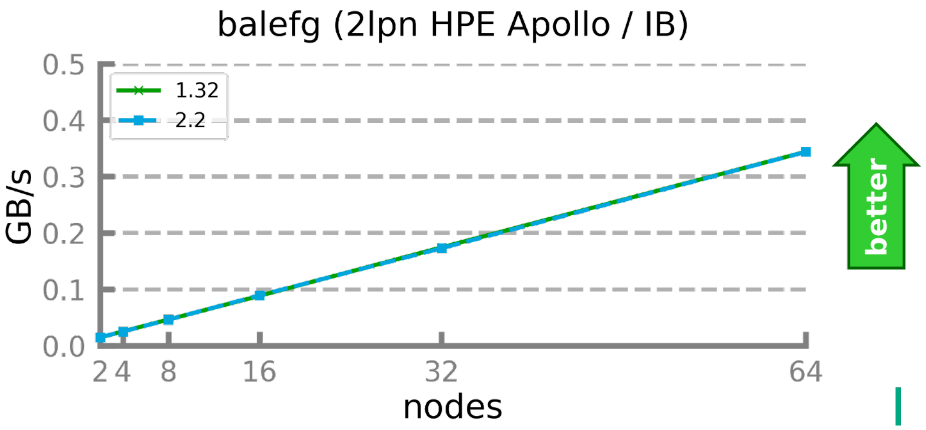
EX/SS11





Apollo/IB





Scalability Since 1.32

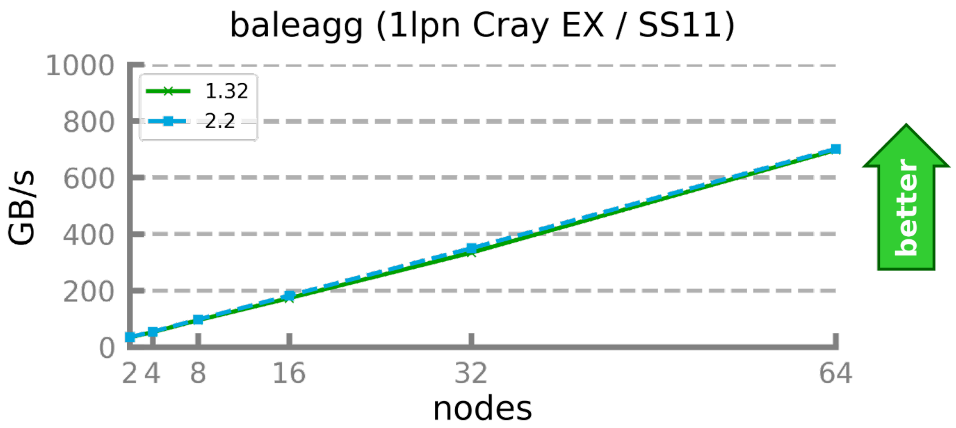
Bale IndexGather – Aggregated Performance

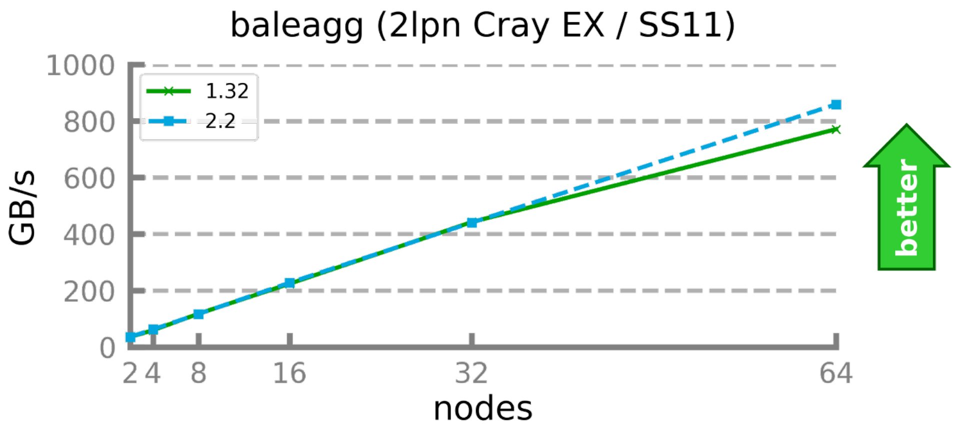
- Similar scalability across versions

1 locale-per-node

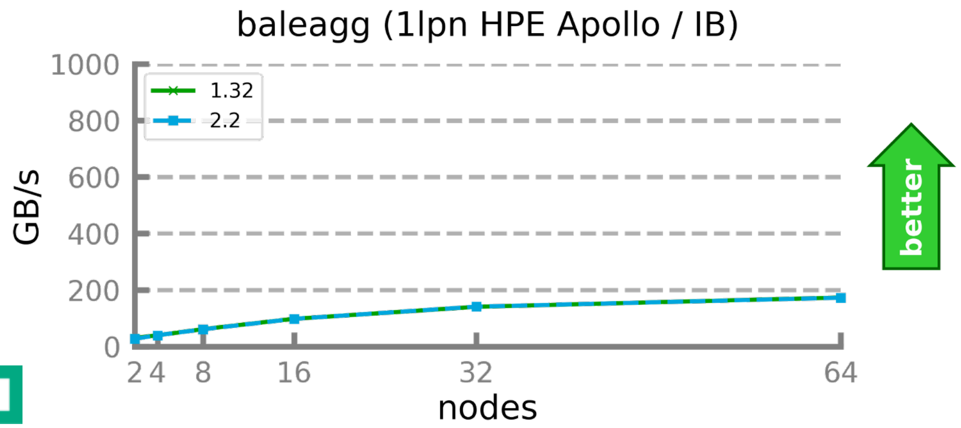
2 locales-per-node

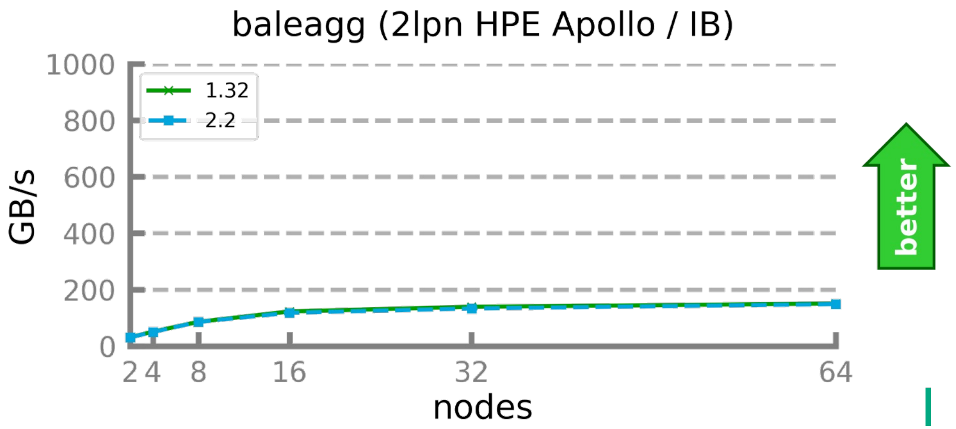
EX/SS11





Apollo/IB





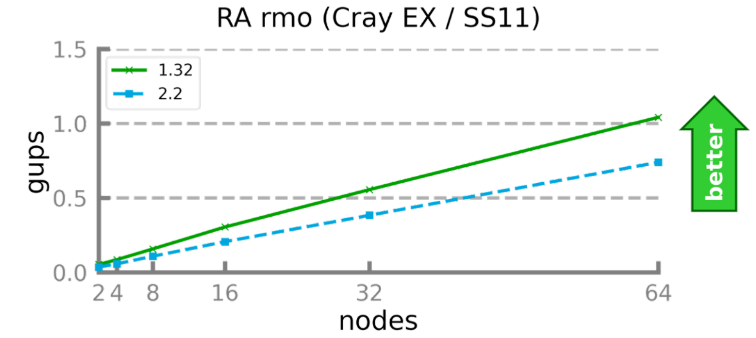
Scalability Since 1.32

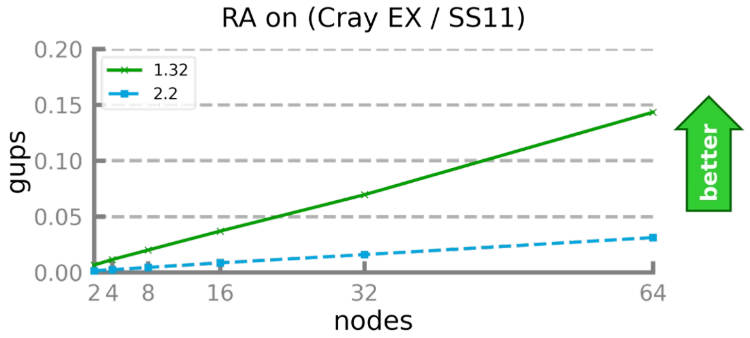
RA Performance

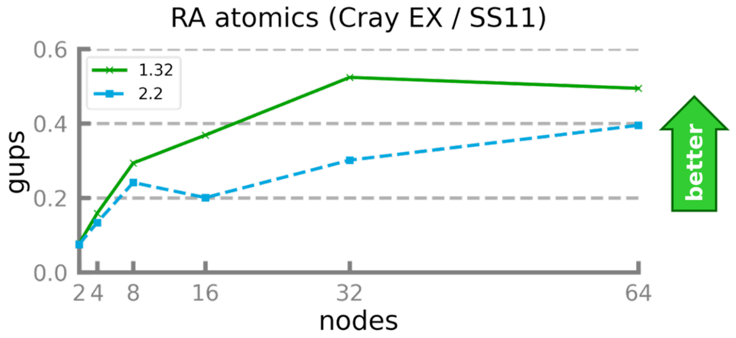
- We see performance regressions for this benchmark on HPE Cray EX/SS11 between 1.32 and 2.2
  - (see the following section for details)

rmo | **on** | **atomics**

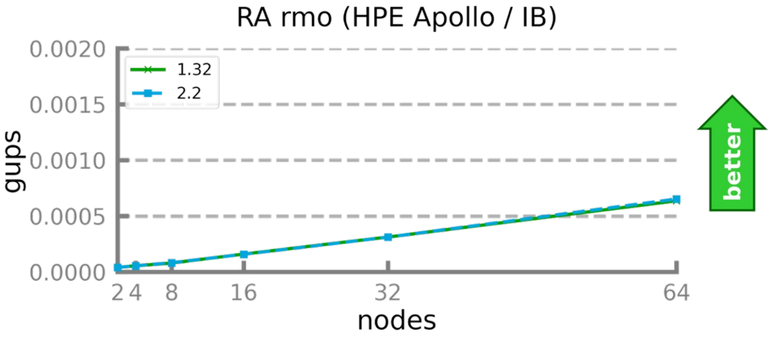
EX/SS11

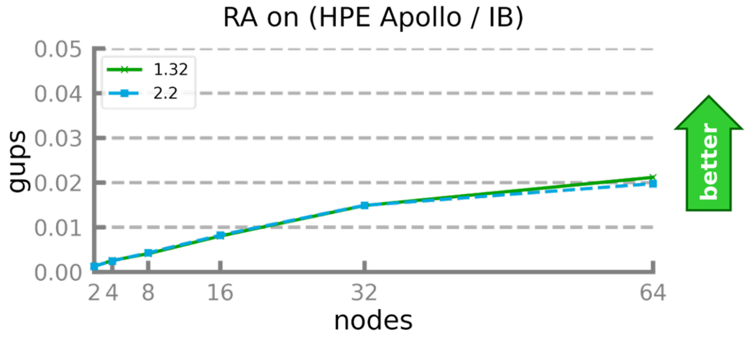


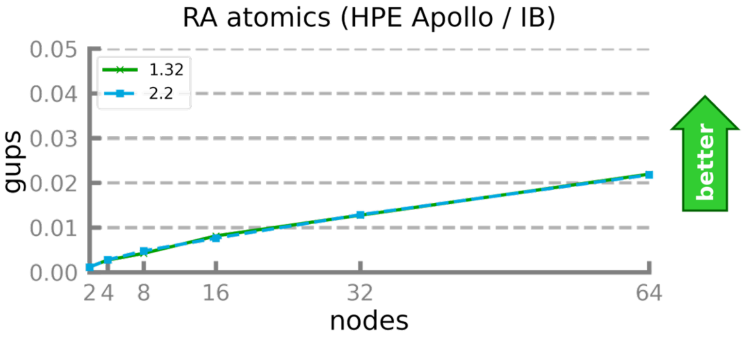




Apollo/IB



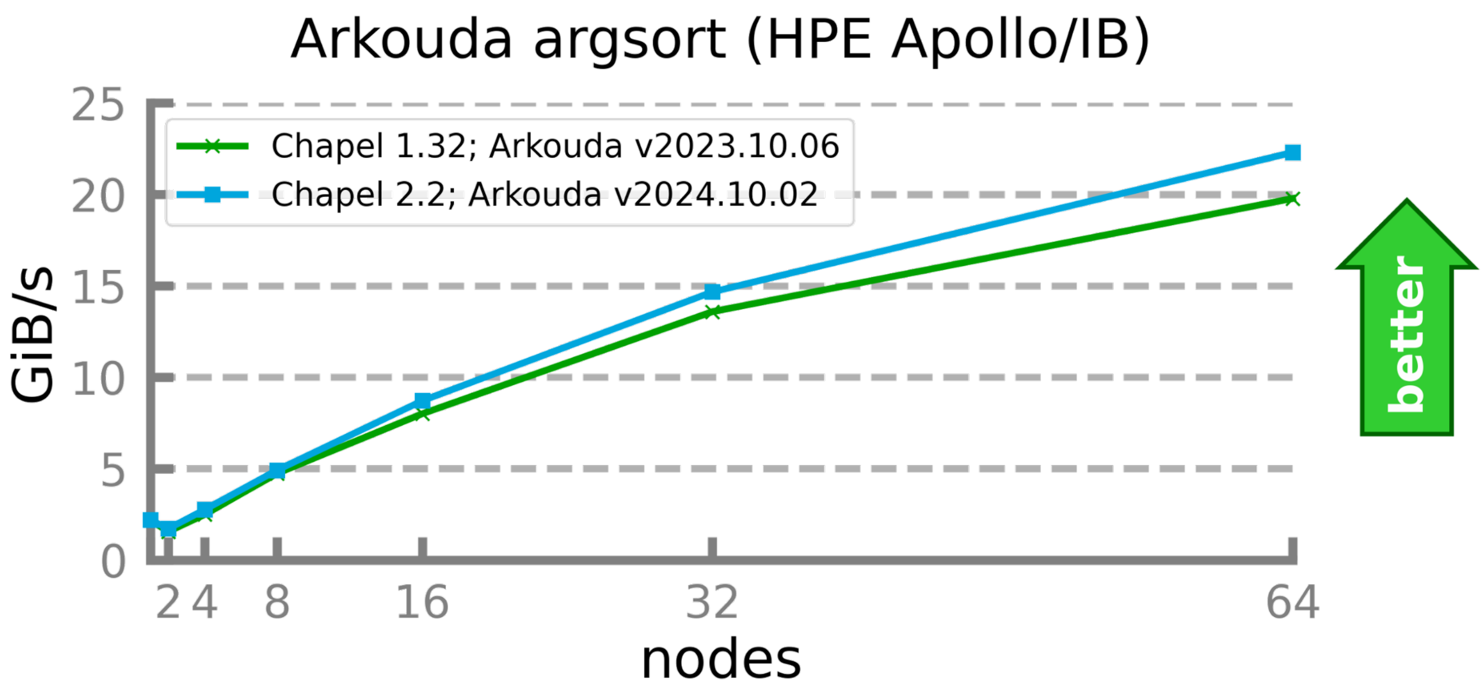




Scalability Since 1.32

Arkouda Argsort

- For Arkouda we only gathered results on the HPE Apollo / InfiniBand machine
- Performance has improved in 2.2 (0-13% higher GiB/s depending on node count)



Performance Regressions and Resolutions

Performance Regressions and Resolutions

ra-rmo

Background:

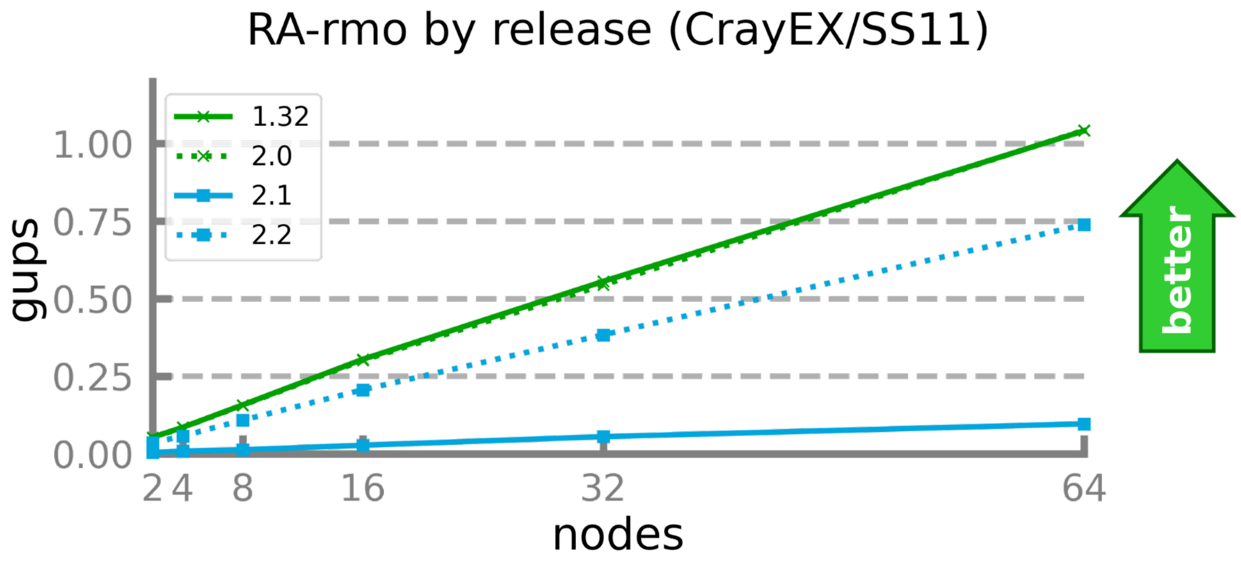
- In Chapel 2.1, we incorrectly added a write-after-write ordering requirement
  - But compiler emits blocking PUTs
  - Software cache uses non-blocking PUTs, but enforces ordering to the same address
- Blocking PUTs were inadvertently non-blocking
  - Could lead to hangs due to lack of progress
  - Non-blocking PUTs implemented via blocking PUTs

This Effort:

- Removed write-after-write for 2.2

Next Steps:

- Improved non-blocking PUTs will be available in 2.3



Performance Regressions and Resolutions

ra-on

Background:

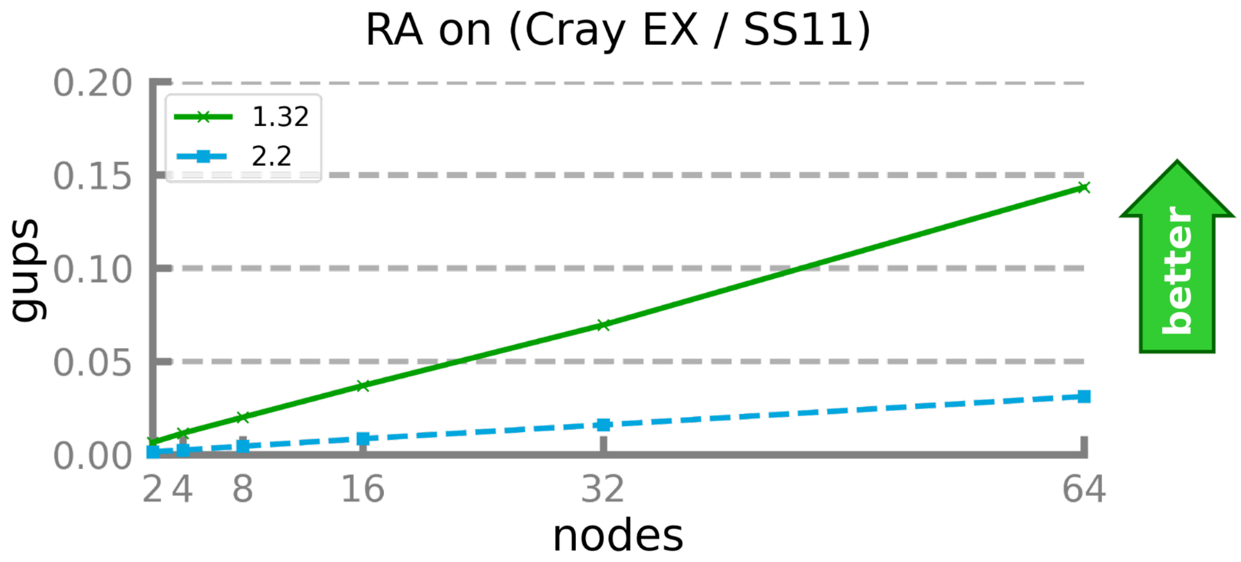
- After a blocking ‘on’, a flag is PUT to the sender indicating that the ‘on’ is complete
- In Chapel 2.1, this PUT was inadvertently non-blocking
  - Could lead to a hang
- Making it blocking reduced performance

Status:

- Resolution is a work-in-progress

Next Steps:

- AM handler must progress transmit endpoint
- Full-scale non-blocking PUT probably too complicated and has too much overhead
- Should be fixed in Chapel 2.3



Performance Regressions and Resolutions

ISx

Background:

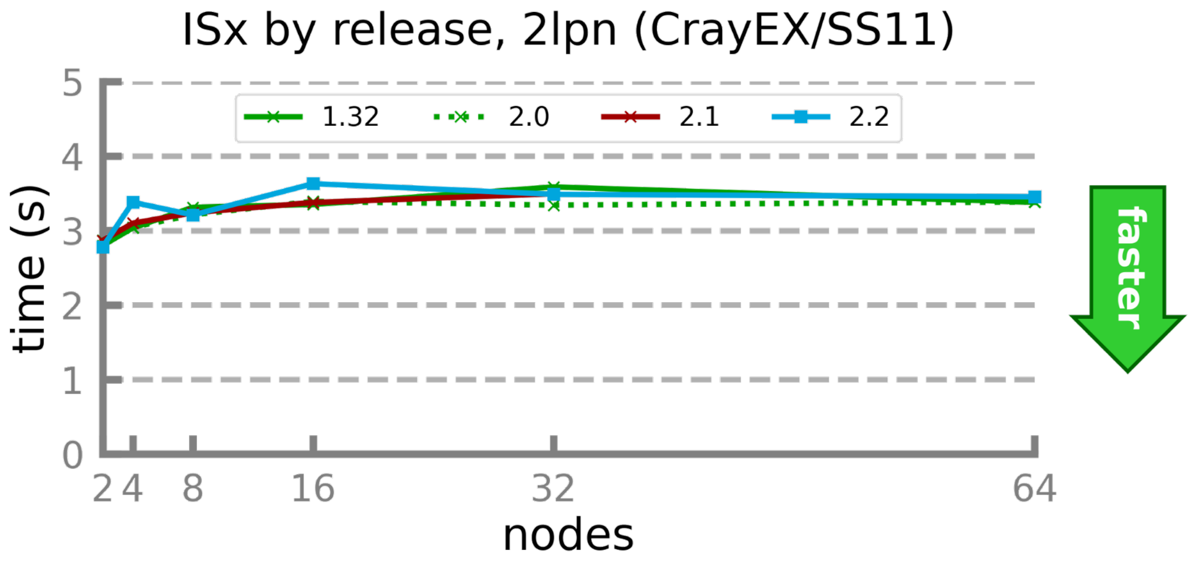
- In Chapel 2.1, ISx would hang at 64 nodes
- Caused by use of FI_DELIVERY_COMPLETE
  - Required by libfabric to force visibility of previous PUTs
- ‘cxi’ provider (SS11) does not implement it

This Effort:

- Resolved hanging behavior in Chapel 2.2
  - By removing use of FI_DELIVERY_COMPLETE

Next Steps:

- Need different mechanism to force visibility
  - Probably cxi-specific
  - Should be addressed in Chapel 2.3



New Optimizations

New Optimizations

- Domain Localization
- Optimizing Array Moves
- Array View Elision
- Optimizing Stencil Distributions

Domain Localization

Domain Localization
Background

- Sometimes, it can be useful to make a local copy of a remote, single-locale array:

```
var A: [1..10] real = computeA();
on Locales[1] {
  const B = A;
  // compute with B here
}
```

- Intuitively, computations on ‘B’ should be completely local / free of communication
- However, in practice, computing with ‘B’ will communicate back to A’s locale to reference its domain
  - This has been surprising and frustrating to end-users
- A common workaround is to also make a local copy of the domain (but this feels annoying):

```
var A: [1..10] real = computeA();
on Locales[1] {
  const D = A.domain,
        B: [D] A.eltType = A;
  // compute with B here
}
```

Domain Localization
Rationale for Status Quo

- Original example:

```
var A: [1..10] real = computeA();
on Locales[1] {
  const B = A;
  // compute with B here
}
```

- In general, this behavior is necessary in case the domain changes:

```
var D = {1..10, 1..10},
    A: [D] real = computeA();
on Locales[1] {
  var B = A;                // B is also declared over 'D'
  D = {1..20, 1..20};       // both 'A' and 'B' need to be re-allocated
  // computing with B requires knowing D's bounds
}
```

- However, when the domain doesn't change, communicating to read it for each op shouldn't be necessary

Domain Localization

This Effort and Status

This Effort:

- When an array's domain is sufficiently 'const', the compiler now localizes it along with the array:
  - When the domain is anonymous or declared 'const', we know it cannot change
  - When the array copy is 'const', we know the domain can't change during the copy's lifetime
    - Note: our motivating example meets both conditions since A's domain is anonymous and 'B' is declared 'const' (but either is sufficient)

```
var A: [1..10] real = computeA();
on Locales[1] {
  const B = A;

  // compute with B here
}
```

optimized similarly to the user-level rewrite

```
var A: [1..10] real = computeA();
  on Locales[1] {
    const D = A.domain,
          B: [D] A.eltType = A;
    // compute with B here
  }
```

Status:

- Optimization was available in Chapel 2.1, but off by default (enabled by compiling with '-slocalizeConstDomains')
- Optimization was enabled by default in Chapel 2.2

Domain Localization
Impact

- Computation on localized arrays now incurs no array-driven communication, enabling ‘local’ block usage
- Degree of impact can be arbitrarily large depending on the number of ops performed on the array

```
var A: [1..n, 1..n] real;
on Locales[1] {
  const B = A;
  for i in 1..iters do
    B += 1.0;
}
```

	unoptimized			optimized
	0 iters	100 iters	10,000 iters	any # of iters
gets	25	1125	110,025	15
active messages	1	1	1	0

- For the main kernel in a user-motivated primes sieve computation (problem size 50,000,000,000):

unoptimized:

locale	get	put	execute_on	execute_on_nb
0	8904	6702	0	1113
1-3	14098	0	2226	0

optimized:

locale	get	execute_on	execute_on_nb
0	2226	0	1113
1-3	7420	742	0

Domain Localization
Next Steps

Next Steps:

- Look into reducing the amount of communication used to localize domains, to ensure it's minimal
  - Particularly for sparse domains which currently require O(*nnz*) remote gets to localize, but should be O(1)
- Consider array implementations that need fewer references to their domains
  - e.g., for dense, rectangular cases, consider storing the bounds directly in the array's descriptor?
- Explore opportunities to strengthen the optimization:
  - Add compiler analysis to cover more cases where a domain is sufficiently invariant? (e.g., def-use analysis)
  - When multiple arrays sharing a domain are localized, investigate sharing the localized domain as well?

```
const D = {1..10};
var A, B, C: [D] real;
on Locales[1] {
  var X = A,      // today, this will create a copy of 'D' per array, but one copy would suffice for X, Y, and Z
      Y = B,
      Z = C;
}
```

Optimizing Array Moves

Optimizing Array Moves
Background

- Array types in Chapel include the domain as a runtime component to represent the array shape

```
var A: [1..n] int;    // '[1..n] int' is a type, even though 'n' can vary at runtime
// the above is shorthand for the following:
const MyDomain = {1..n};
var B: [MyDomain] int;    // '[MyDomain] int' is a type
```

- In this context, the specific domain variable is important, not just the index set
  - Why? Because assigning to a domain can resize the arrays declared over it
- As a result, returning an array can result in an implicit conversion, to match a declared return type

```
config const n = 1_000_000;
var D = {1..n};
proc createArray(): [D] real {
  var MyArray: [1..n] real = ...;
  return MyArray;    // here, compiler must convert from the type '[1..n] real' to the type '[D] real'
}
```

- Historically, this pattern has led to allocating a new array to implement the implicit conversion
  - Could even lead to out-of-memory errors when the arrays are sufficiently large

Optimizing Array Moves
This Effort

- Optimized the implementation of such array moves with equivalent but different domains
- For this initial effort, limited the optimization to a common case:
  - Default rectangular arrays that aren’t arrays of arrays

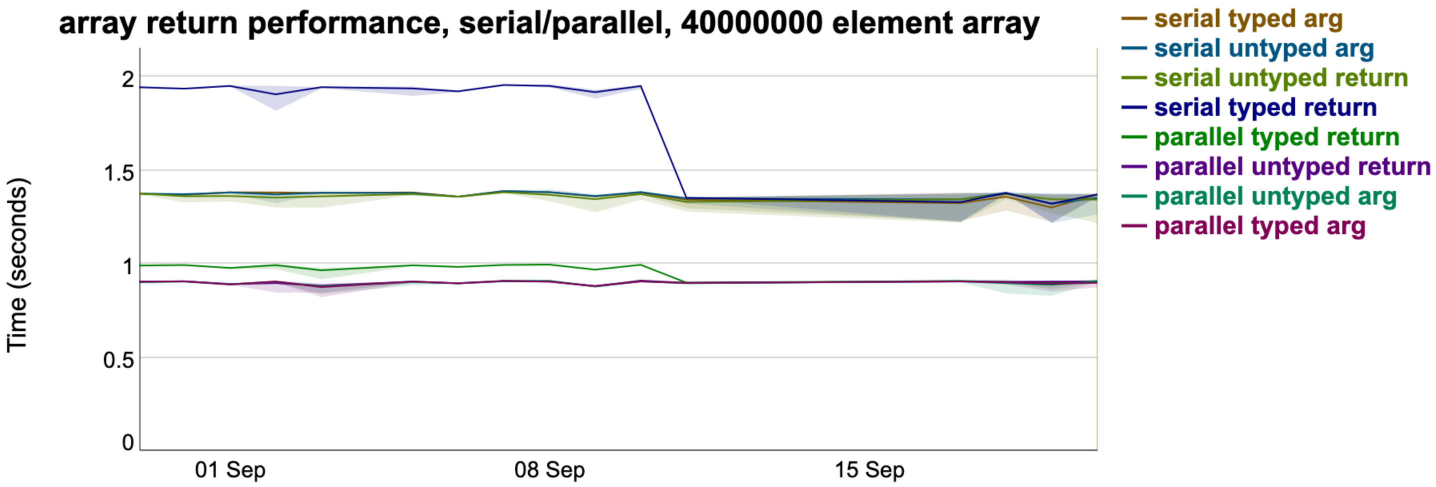
- Avoids two array allocations in the below code:

```
proc createArray(): [D] real {
  var MyArray: [1..n] real = …;    // note the difference from the declared return type

  return MyArray;    // Array allocation for moving ‘[1..n] real’ to ‘[D] real’ is avoided
}

var OtherArray: [1..n] real = createArray();    // Array allocation for moving ‘[D] real’ to ‘[1..n] real’ is avoided
```

Optimizing Array Moves

Impact and Next Steps

Impact: Improved performance and reduced one source of out-of-memory errors



Next Steps:

- Implement the optimization for other array types, especially the distributed arrays Block, Cyclic, and Stencil
- Get the optimization working for arrays of arrays

Array View Elision

Array View Elision

Background

- Array views are a kind of array that refers to another array
  - A common example is an array slice:

```
var A: [1..10] int;
ref ACenter = A[3..8];
```

  - All arrays, including array views, have a consistent interface:

```
writeln(ACenter.size);        // prints "6"
writeln(ACenter.domain);      // prints "{3..8}"
ACenter = 1;                  // sets all elements at the "center" of A to 1
```

- A common pattern in Chapel is to copy between chunks of two arrays
  - This is implemented with array views:

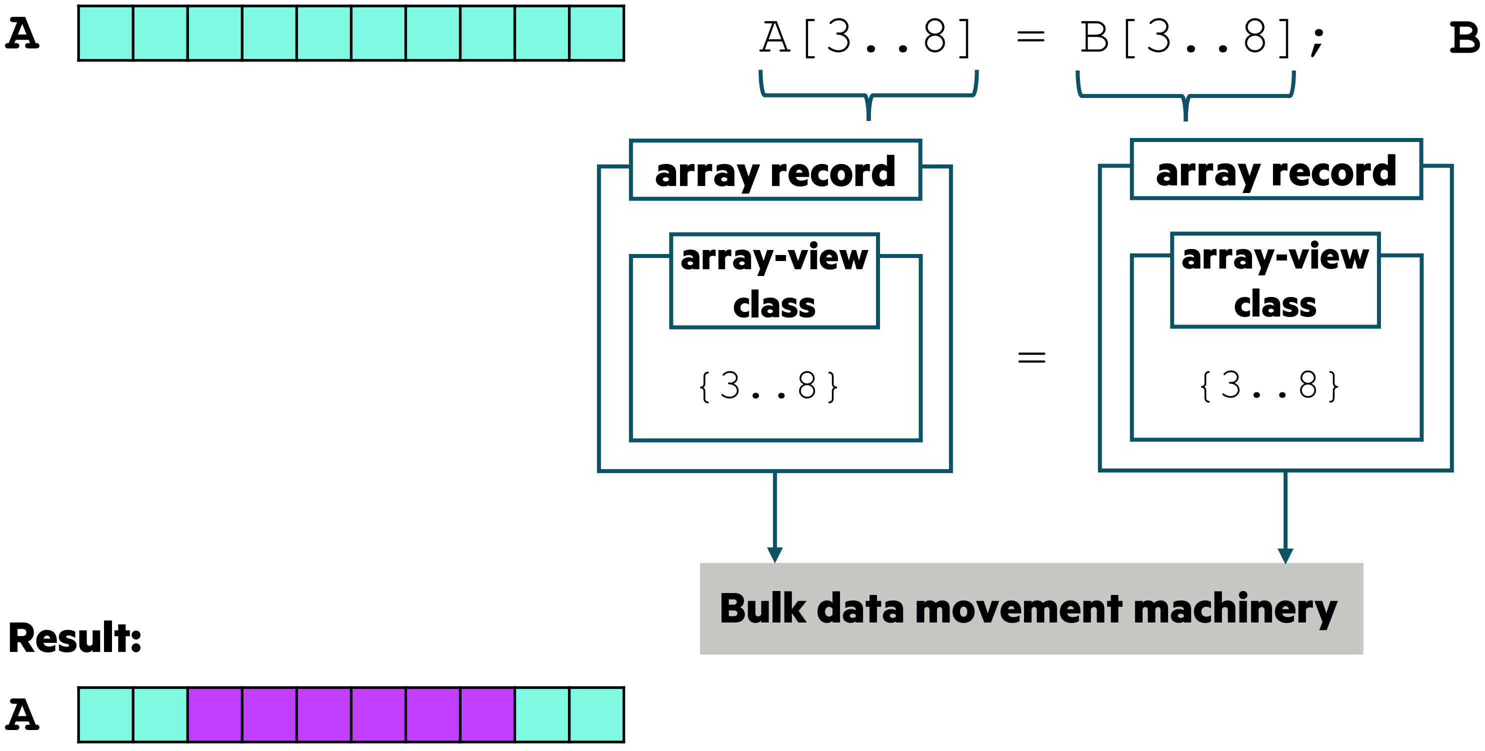
```
var A, B: [1..10] int;
A[3..8] = B[3..8];
```

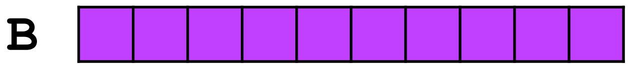
Array View Elision

Background

- The common pattern of copying between two slices had a lot of overhead

```
var A, B: [1..10] int;
```





The following are created for each slice:

- a record instance for the array interface
- a class instance for the array view
- a domain for the slicing expression

This results in:

- 4 dynamic allocations
- 4 domain/array initializations

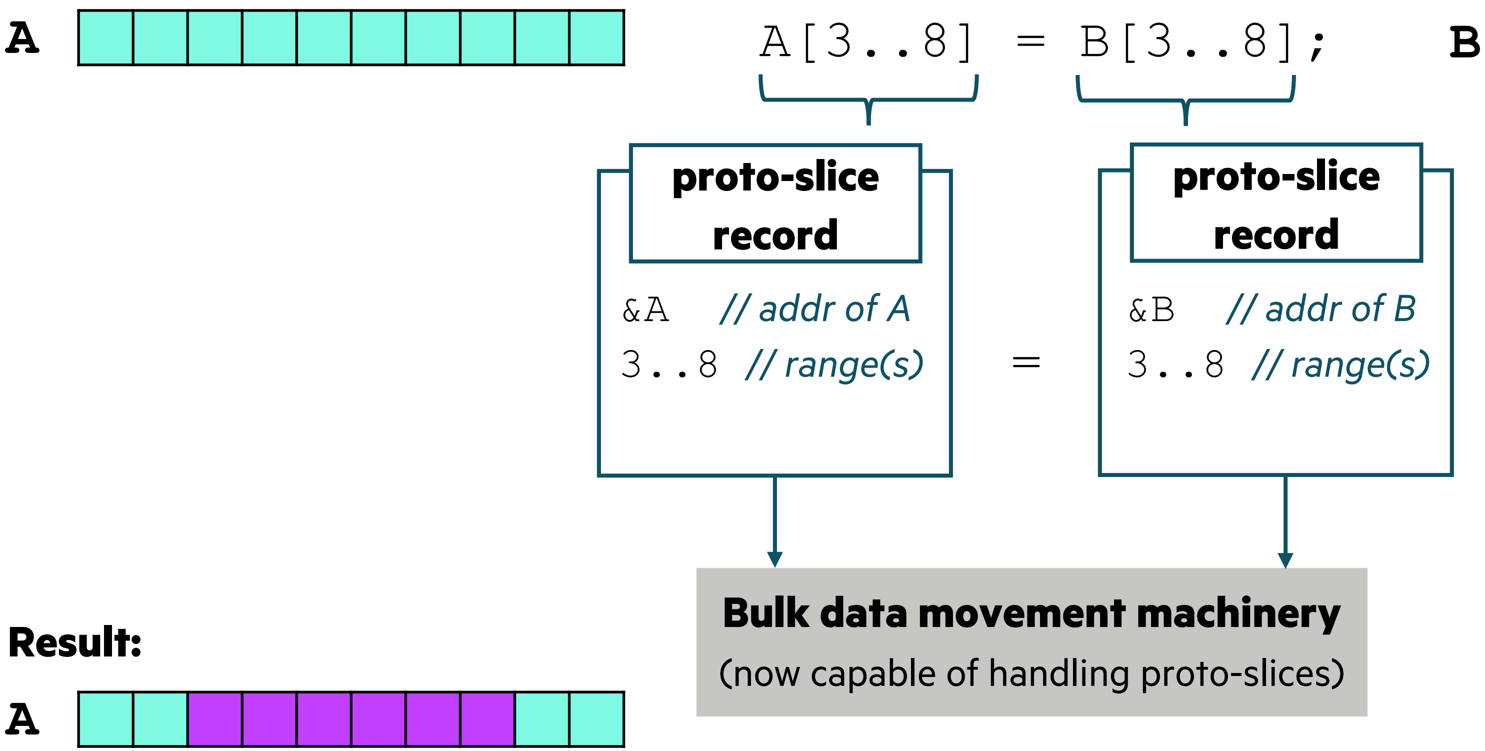
These costs can impact performance with small transfers

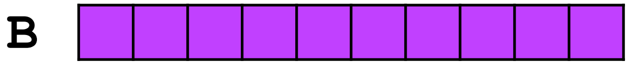
Array View Elision

This Effort

- With Chapel 2.2, the compiler detects this common pattern and optimizes it:

```
var A, B: [1..10] int;
```





A “proto-slice” record stores:

- the original array's address
- a (tuple of) range(s)

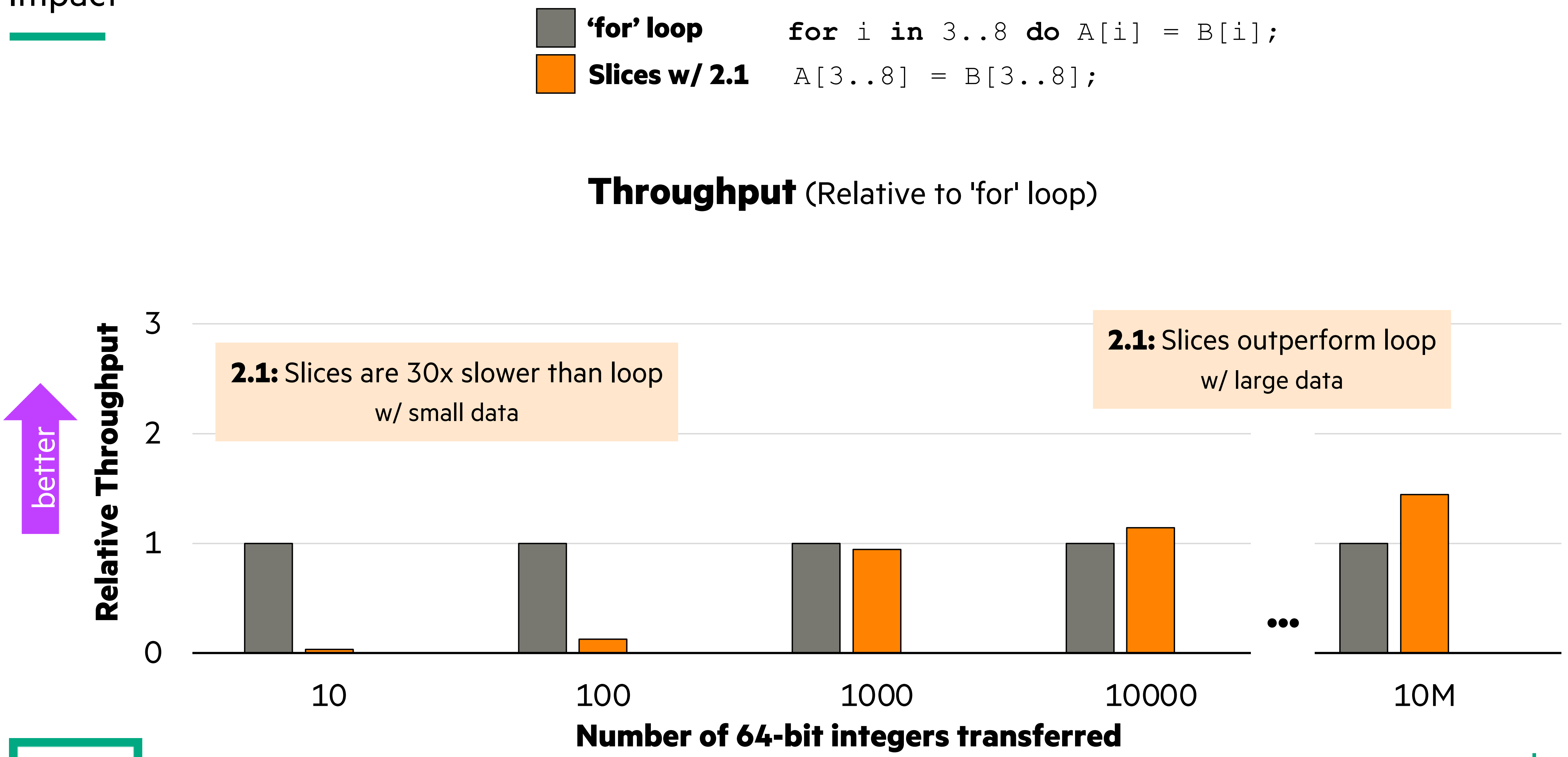
This results in:

- no dynamic allocations
- much faster initialization

Significantly reduced overhead for a common operation

Array View Elision

Impact

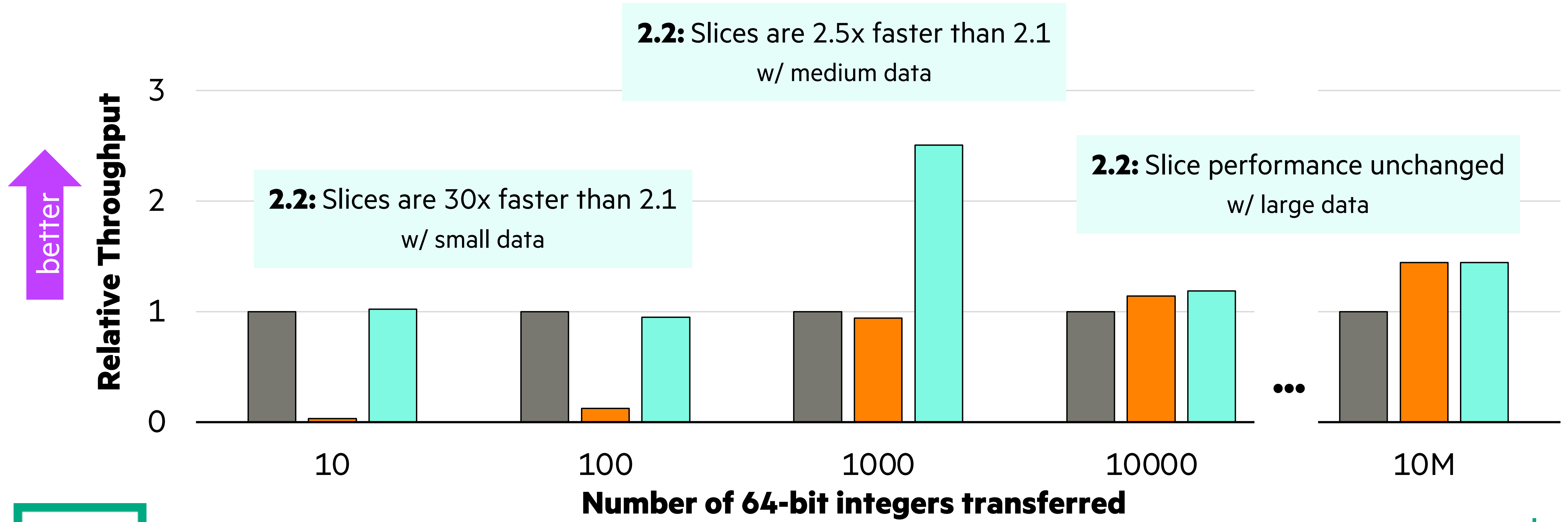


Array View Elision

Impact

'for' loop	`for i in 3..8 do A[i] = B[i];`
Slices w/ 2.1	`A[3..8] = B[3..8];`
Slices w/ 2.2	`A[3..8] = B[3..8];`

Throughput (Relative to 'for' loop)



Array View Elision

Status and Next Steps

Status: Assignments between same types of views are supported. e.g.:

```
A[3..8] = B[3..8];                    // 1D slice to slice
A[3..8, 3..8] = B[3..8, 3..8];        // Multi-dimensional slice to slice
A[1, ..] = B[3, ..];                  // 1D rank-change to rank-change
A[..,4,..] = B[..,2,..];              // Multi-dimensional rank-change to rank-change
```

Next Steps: Array view elision can be expanded to cross-type assignments. e.g.:

```
A[3..8] = C;             // array to slice
D = B[3, ..];            // rank-change to array
A[3..8] = E[4, 3..8];    // rank-change to slice
```

Optimizing Stencil Distributions

Stencil Distribution Performance Improvements

Background: stencilDist's performance has been worse than blockDist for some small stencil codes

This Effort:

- Minimized communication overhead in stencilDist's 'updateFluff' method
- Expanded auto-local-access optimization to optimize array accesses within stencilDist's fluff region

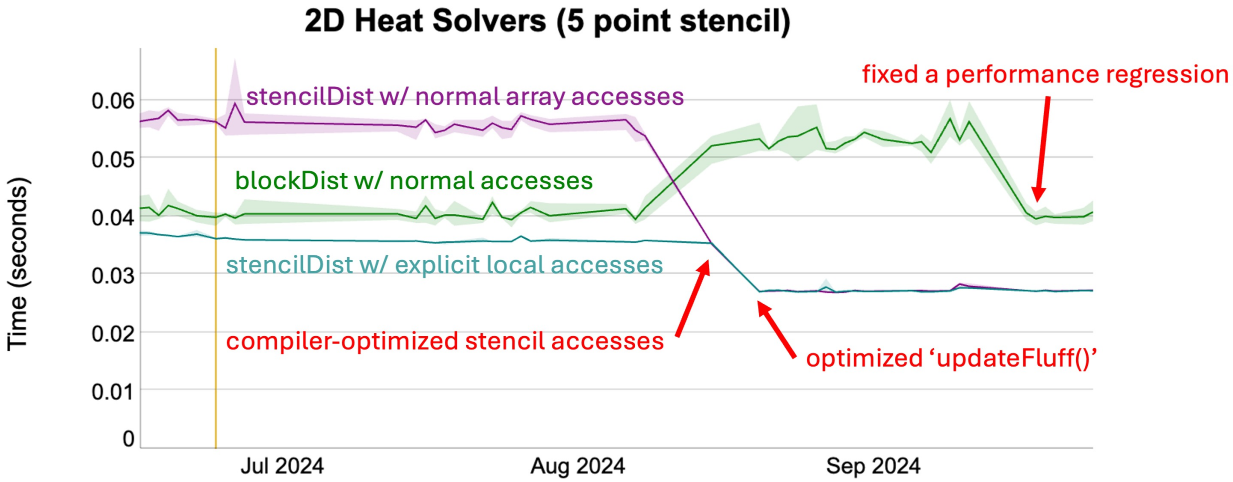
```
forall i in Arr.domain.expand(-1) {     // iterate over the inner portion of the array's domain
  Arr[i] = (Arr[i-1] + Arr[i] + Arr[i+1])/3;
```

Optimized since Chapel 1.23

New optimization in Chapel 2.2

Impact:

- Explicit 'localAccess' unneeded in most stencil codes
- Overall performance of fluff updates is improved
- See 2.2 release announcement for more details



Other Performance Improvements

Other Performance Improvements

For a more complete list of performance changes and improvements in the 2.1 and 2.2 releases, refer to the following section in the CHANGES.md file:

- Performance Optimizations / Improvements

Thank you

https://chapel-lang.org
@ChapelLanguage