# Improvements to Arrays and Domain Maps

Chapel Team, Cray Inc.

Chapel version 1.15

April 6, 2017

# 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.

# Outline

- Improved Array Memory Management
- Arrays Return By Value
- Array Default Argument Intent
- Array Views
- Deprecating Array Alias Operators '=>'
  - Deprecating Array Aliases in Constructor Calls
  - Deprecating Array Aliases in Declarations
- BlockCyclic Improvements
- Array/Domain Shape Methods
- Other Array / Domain Map Improvements

# Improved Array Memory Management

# Array Memory: Overview

- **What is the problem?**
  - Array memory management was incorrect and slow
- **Why do we have this problem?**
  - Original semantics of arrays required reference counting
- **How did we address the problem?**
  - Language changes
  - Leveraging improved semantics
- **What is the result of this work?**
  - Huge reduction in leaks, good performance impact

# Array Memory: Background

- **Array memory management has been problematic**
  - memory leaks
  - performance overhead
- **Largest source of memory leaks in Chapel 1.14**
  - distributed arrays accounted for most leaked data
- **Implementation overheads hurt performance**
  - Benchmarks spent significant time handling array reference counting
    - Supported a 'noRefCount' setting to measure/reduce impact
    - Sometimes helped dramatically, but guaranteed arrays would be leaked
  - Array memory management overheads could be surprising:

```
var size = A.domain.size; // changed reference counts!
```

# Array Memory: How did we get here?

- **Array memory management strategy had two goals:**
  1. **Keep arrays alive past lexical scope**
     - when an array slice/view outlives the original array
     - when arrays are used in ‘begin’ statements
  2. **Minimize array copies**

- **But…**
  - Implementation erred on keeping arrays alive to the point of leaking
  - Reference counting approach was expensive and overly conservative
  - Language definition did not clearly specify array return behavior

# Array Memory: This Effort

- **Changed array behavior to solve these problems**
  - arrays are now returned by value by default
    - see next section for details
  - arrays are now freed when they go out of scope
    - ‘begin’ statements, array slices no longer extend array lifetime
- **New semantics do not require reference counting**
- **Re-implemented array, domain, distribution types to:**
  - remove array reference counting
  - free distributed objects
  - reduce number of special cases in the compiler
- **Improved tuple semantics in support of this effort**
  - see subsequent section for details

# Array Memory: Language Changes

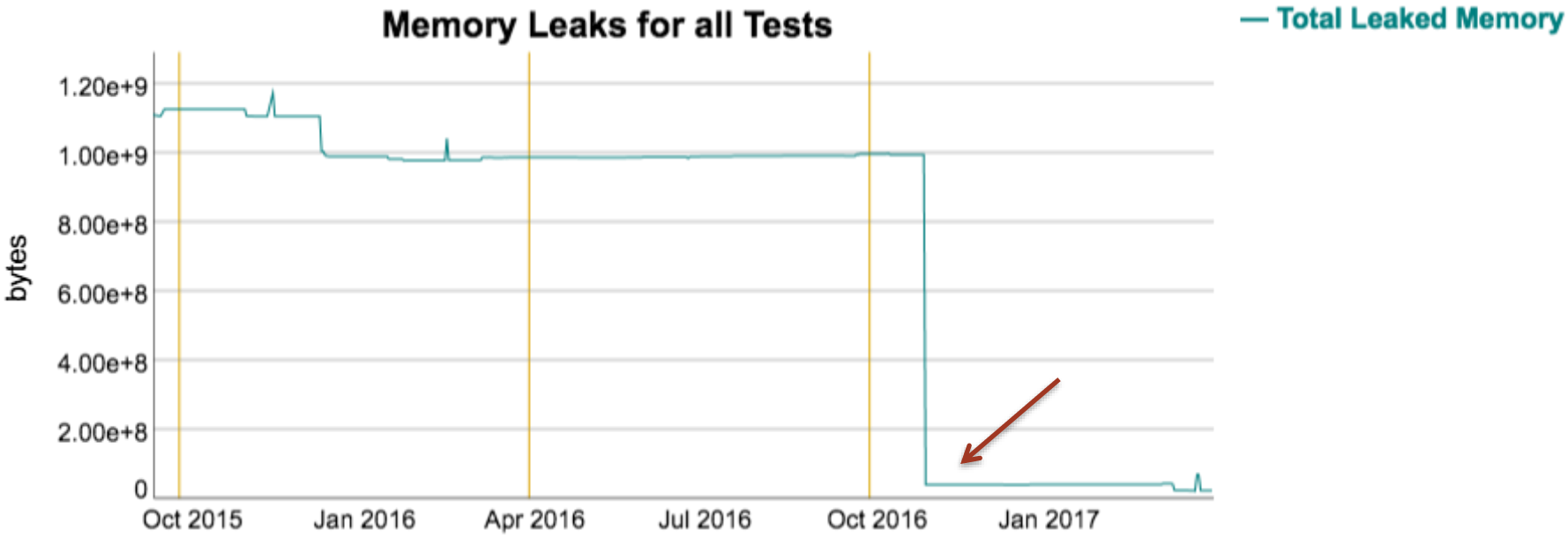
- **Arrays are now destroyed when they go out of scope**
  - 'begin' statements and array slices no longer affect array lifetime

```
proc badBegin() {
  var A: [1..10000] int;
  begin {
    A += 1;
  }
  // User error: A destroyed here at function end, but the begin could still be using it!
}
```

- **A new CHIP describes improved record / array behavior:**
  - CHIP 13: *When Do Record and Array Copies Occur?*

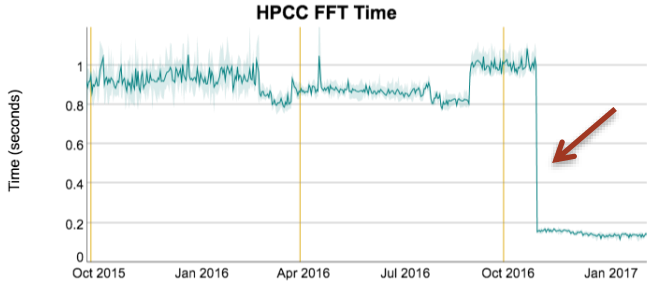
# Array Memory: Impact on Leaks

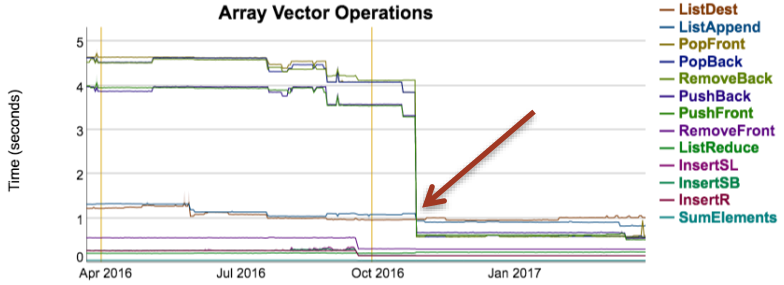
- **Dramatically reduced memory leaks**
  - Closed biggest source of memory leaks
    - e.g., PTRANS benchmark went from leaking 800MB to 0 bytes
  - Distributed arrays no longer leak

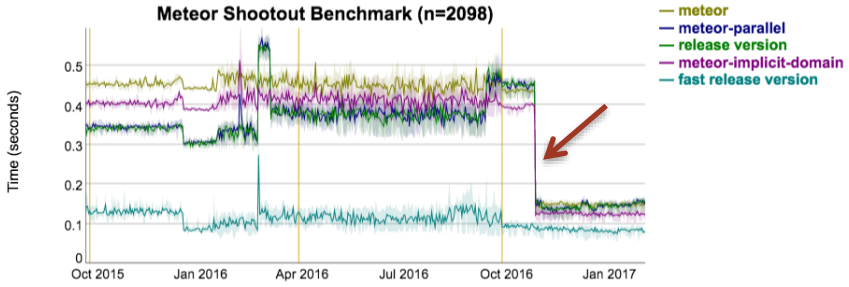


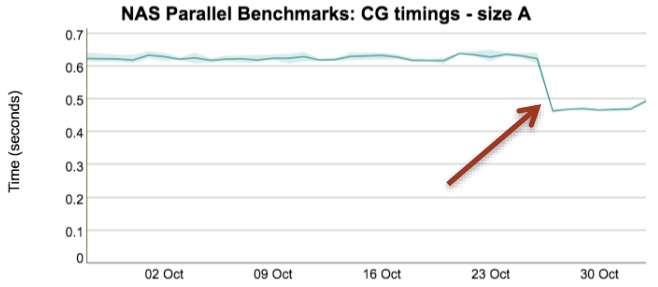
# Array Memory: Impact on Performance

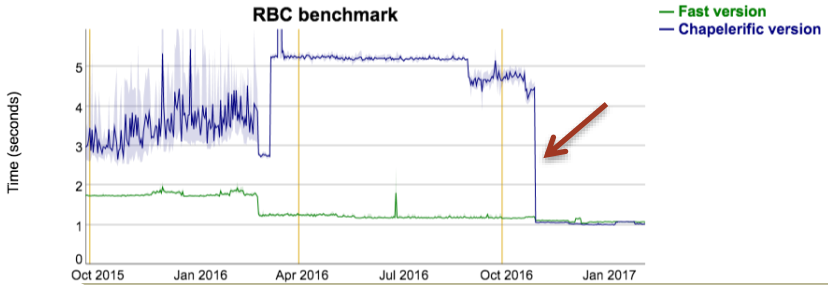
- **Substantial single-locale performance improvements**
  - up to 7x speedup in some cases

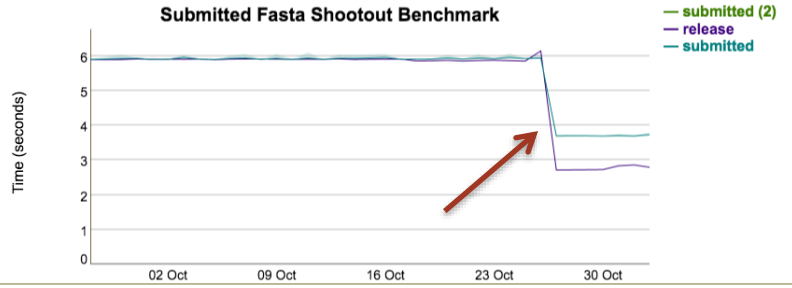


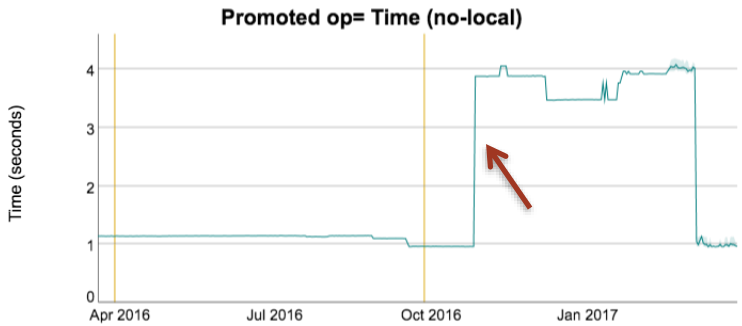



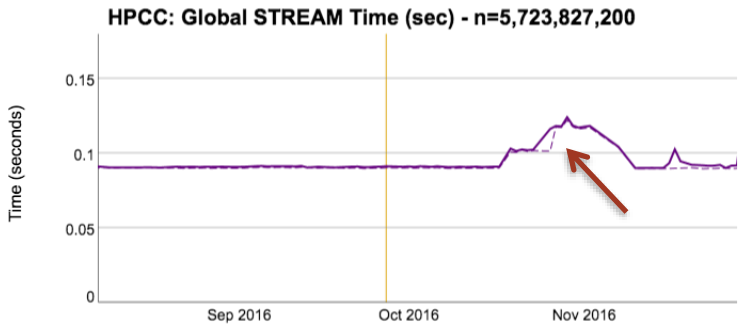
# Array Memory: Impact on Performance

- Caused a few performance regressions
- Follow-on work resolved these regressions
  - Remote Value Forwarding improvements
  - Wide pointer optimization improvement

## --no-local

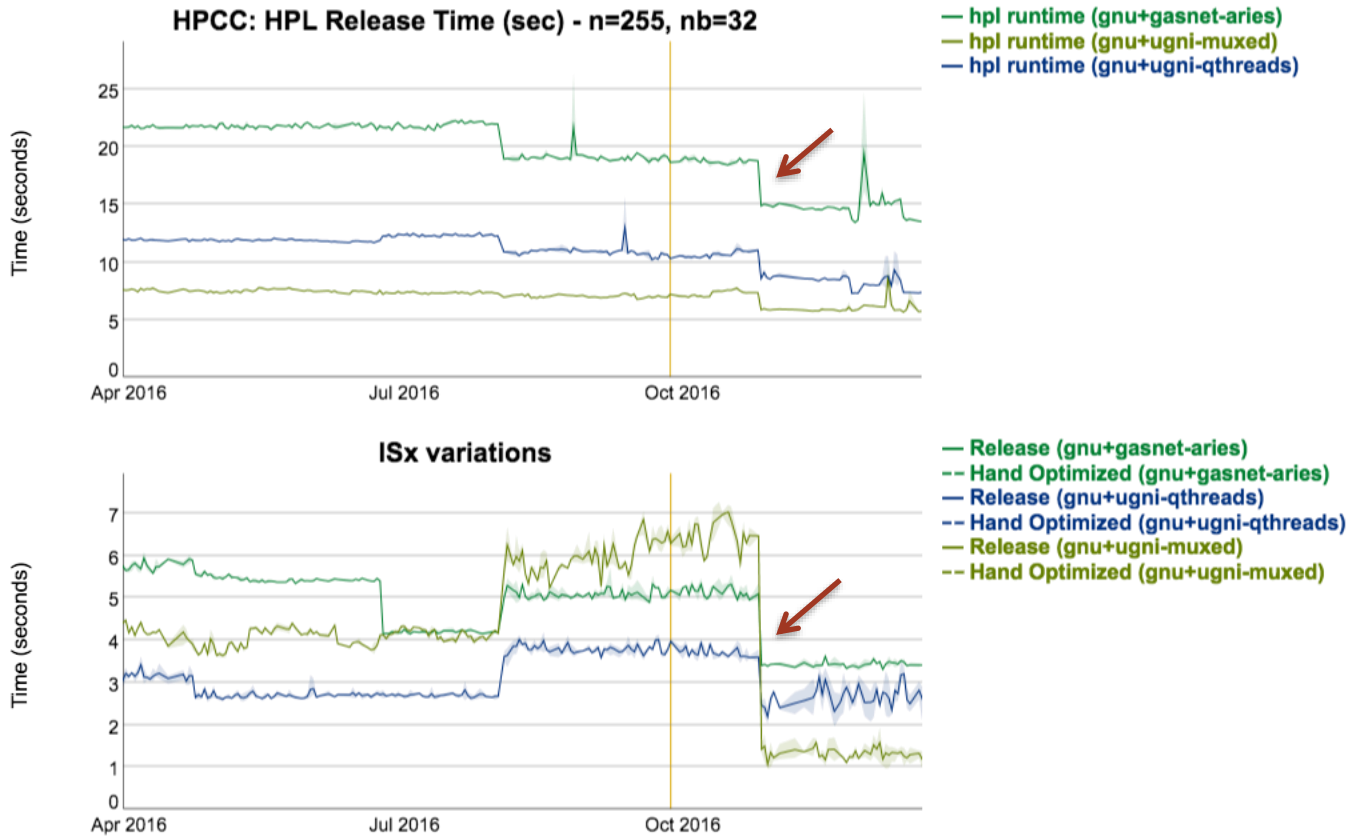



## multi-locale, GASNet-MPI



# Array Memory: Impact on Performance

- **Some big multi-locale performance improvements**
  - up to 6x speedup



# Array Memory: Next Steps (Implementation)

- **Address remaining record memory management issues:**
  - returning a 'ref' intent formal by value should copy but doesn't
  - totally generic member variables can leak
  - variables in an iterator can leak when ‘break’ing from the calling loop
- **Optimize away copies when possible**
  - benefits both arrays and records such as 'bigint'
  - in particular, improve upon
    - assignment or passing by 'in' intent
      …from a call returning an array by value

# Array Memory: Next Steps (Language Design)

- **Develop strategies for tricky design issues:**
  - how to describe when compiler can omit a copy?
  - how can user types opt-in to aggressive copy elimination?
- **Update language specification to describe:**
  - when record/array copies occur
  - tuple semantics in more detail
  - function return is normally by-value

# Arrays Return By Value

# Array Returns: Background and This Effort

**Background:** Arrays historically returned by 'ref' by default

- this design interfered with array memory management improvements

**This Effort:** Changed arrays to return by value by default

- to make them more similar to records
- to simplify the language and its implementation

```
var A: [1..4] int;
proc f() {
  return A; // new in 1.15: return by value
}
ref B = f();
B = 1;
writeln(A);
// printed 1 1 1 1 historically
// prints 0 0 0 0 after this work
```

# Array Returns: Compatibility

- **When old behavior is desired, use 'ref' return intent**
  - should result in behavior that's backwards-compatible with 1.14

```
var A: [1..4] int;
proc f() ref { // explicit ref return
  return A;
}
ref B = f();
B = 1;
writeln(A);
// prints 1 1 1 1
```

# Array Returns: Next Steps

- **Describe array return behavior in language specification**
- **Revisit l-value rules for arrays**
  - if f() returned a record by value, this would be an error:

```
ref B = f();
```

…but it is currently allowed when f() returns an array

# Array Default Argument Intent

# Array Intent: Background

- **Historically, the default intent for arrays was 'ref'**
  - designed as a convenience for programmers
  - avoids surprising programmers used to modifying array formals
- **This design interacted poorly with return intent overloads**
  - return intent overloads allow different behavior on read and write
    - e.g. writing the “zero” values of a sparse array is an error
    - e.g. reading the “zero” values of a sparse array is fine
  - combining this with 'ref' default intent created surprising behavior
    - especially with arrays-of-arrays
- **Not just a matter of accurate const-checking errors**
  - problems originally discovered studying miniMD performance

# Array Intent: Motivation

- **Consider this example with a sparse array of int:**

```
var dense = {1..10};
var sps: sparse subdomain(dense); // domain is initially empty (all zeroes)

var A: [sps] int;

writeln(A[3]); // outputs 0, the "zero" value
```

# Array Intent: Motivation

- **Behavior changes for an array of arrays:**

```
var dense = {1..10};
var sps: sparse subdomain(dense);

var A: [sps] [1..5] int;

writeln(A[3]); // surprising: halts
               // attempting to assign a 'zero' value in a sparse array: (3)
```

- **What's happening in this example?**
  - writeln() takes its arguments by default intent
  - default intent for an array is 'ref'

  ➔ writeln() appears to the array implementation to set its argument
    - setting a sparse array's "zero" values via indexing is not permitted

# Array Intent: This Effort

- **Changed the default intent for arrays…**
  …to 'ref' if the formal argument is modified in the function body
  …to 'const ref' if not

```
proc setElementOne(x) {
  // x is modified, so x has 'ref' intent
  x[1] = 1;
}

var A:[1..10] int;
setElementOne(A);
```

```
proc getElementOne(y) {
  // y is not modified,
  // so y has 'const ref' intent
  var tmp = y[1];
}
var B:[1..10] int;
getElementOne(B);
```

- **Fixed related bugs in the implementation**

# Array Intent: Resulting Behavior

- **Motivating cases now work as you'd expect:**

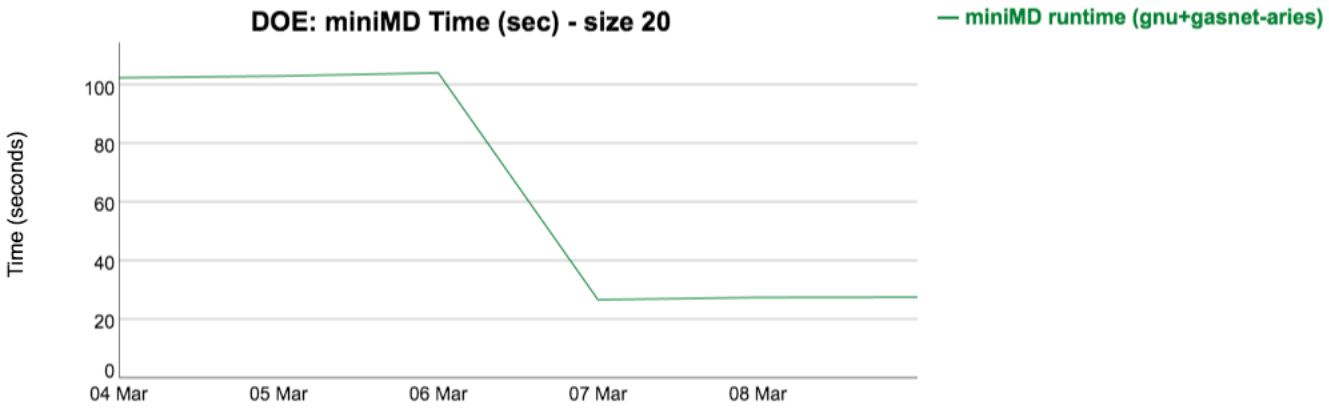
```
var dense = {1..10};
var sps: sparse subdomain(dense);

var A: [sps] [1..5] int;

writeln(A[3]); // prints A[3]
```

- **Why does this now work?**
  - writeln() still takes its arguments by default intent
  - because it only reads its args, the default intent for arrays is 'const ref'

  ➔ writeln now calls the array's read accessor
    - reading a sparse array's "zero" values is fine

# Array Intent: Impact

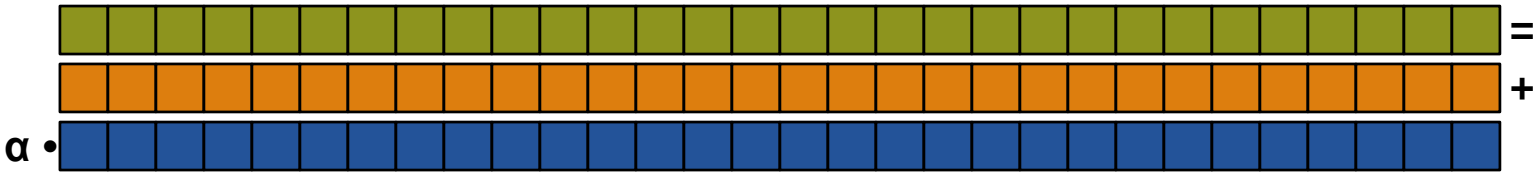
- **Reduced a source of bugs and confusion**
  - makes the language more consistent and less surprising
  - still meets the original array intent design goals:
    - simple programs can be written without argument intents
    - avoids surprising programmers accustomed to modifying array formals
- **Led to about 4x speedup for miniMD on 16 nodes**
  - StencilDist uses return-intent overloads to return from a cache
  - This effort enabled the cache for arrays-of-array stencils, as in MiniMD



# Array Views

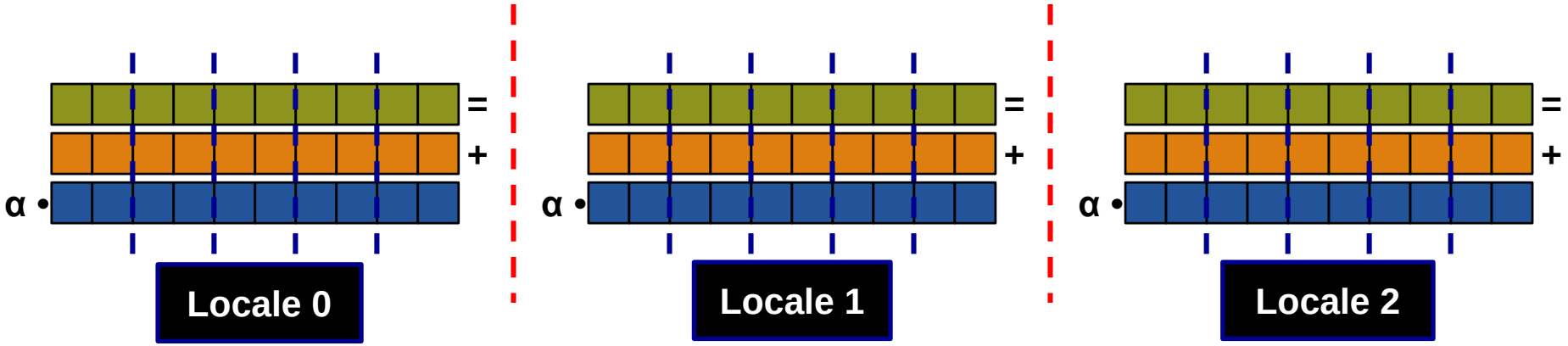
# Array Views: Background (Domain Maps)

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation…




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

…to the target locales’ memory and processors:



# Domain Map Definitions

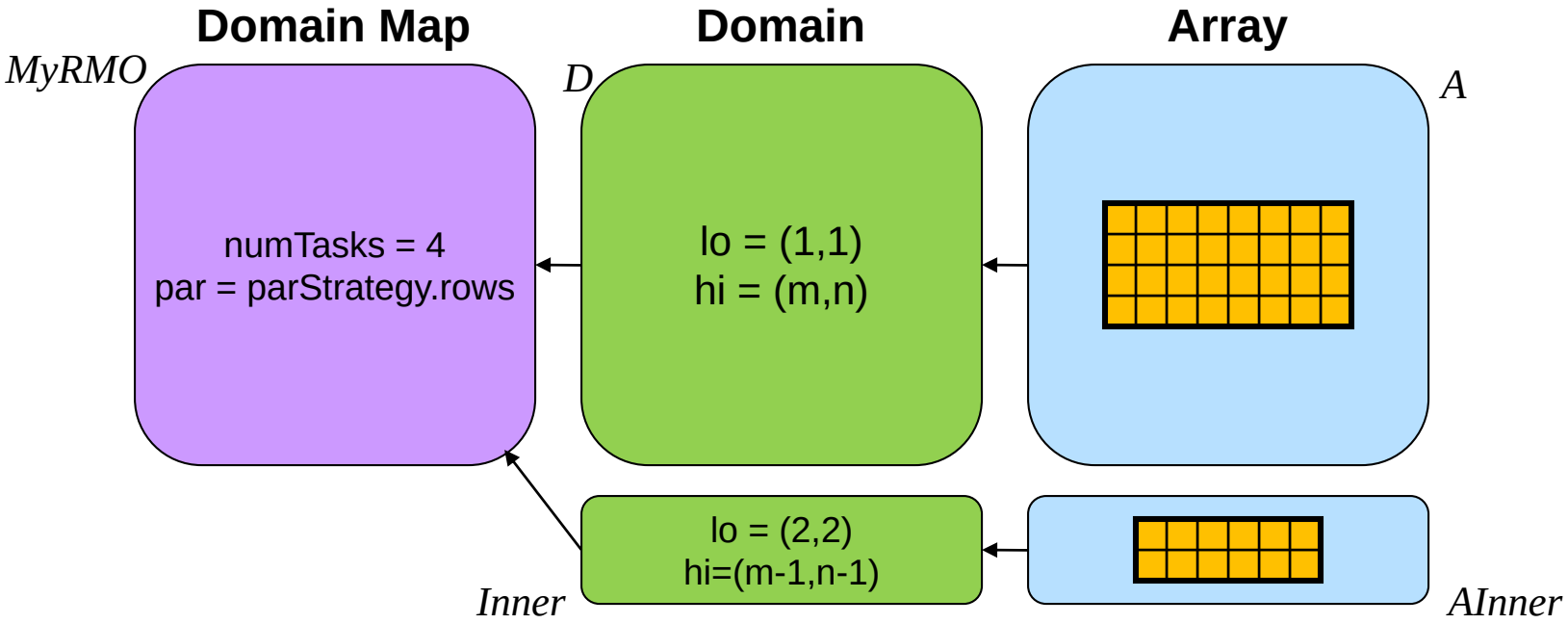
## Domain Maps:

- recipes for implementing domains and their arrays
- come in two flavors:
  1. layouts:
     - target a single locale
     - specify memory layout
  2. distributions:
     - target multiple locales
     - specify distribution of indices to locales
     - typically implemented using a layout within the locale
- implemented using three classes (“descriptors”):
  1. the domain map object itself
  2. an object for each domain implemented using that domain map
  3. an object for each array implemented using that domain

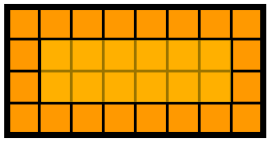
# Domain Maps: Descriptor Interfaces

- **Domain Map Interfaces**
  - domain map objects must specify:
    - which locale owns a given index
    - …
    - how to create domains
  - domain objects must specify:
    - how to iterate over indices
    - how to intersect with other domains
    - how to test for membership
    - …
    - how to create arrays
  - array objects must specify:
    - how to access the array's elements
    - how to iterate over the array's elements
    - …

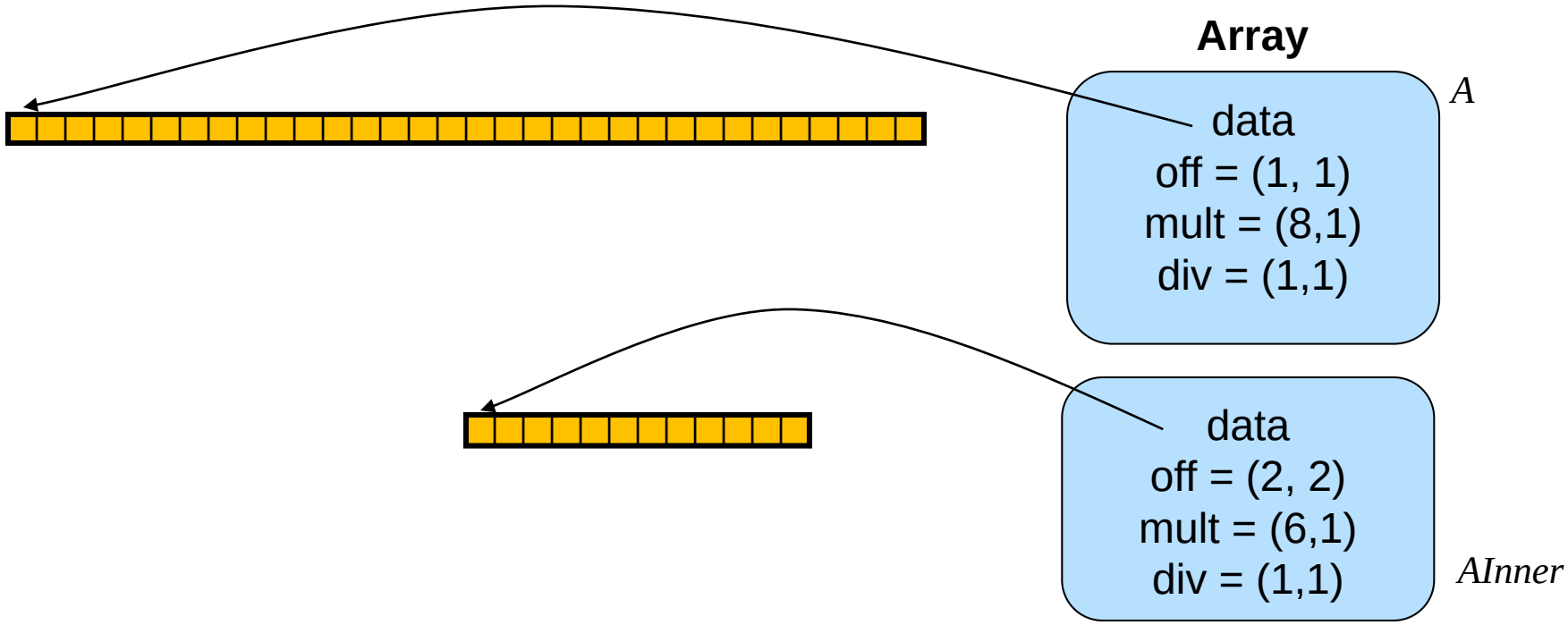
# Sample Descriptors



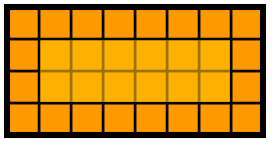

```
const MyRMO = new dmap(new RMO(here.numCores, parStrategy.rows));

const D = {1..m, 1..n} dmapped MyRMO,
      Inner = D[2..m-1, 2..n-1];

var A: [D] real,
    AInner: [Inner] real;
```



# Array Descriptors




```
const MyRMO = new dmap(new RMO(here.numCores, parStrategy.rows));

const D = {1..m, 1..n} dmapped MyRMO,
      Inner = D[2..m-1, 2..n-1];

var A: [D] real,
    AInner: [Inner] real;
```



# Three “Array View” Operations

## Slicing:

```
…A[lo1..#b, lo2..#b]…   // refer to a b x b block starting at (lo1, lo2)
…A[.., i..i]…           // refer to the ith column of A (as a 2D array)
```

## Rank Change:

```
…A[.., i]…              // refer to the ith column of A as a 1D array
```

## Reindexing:

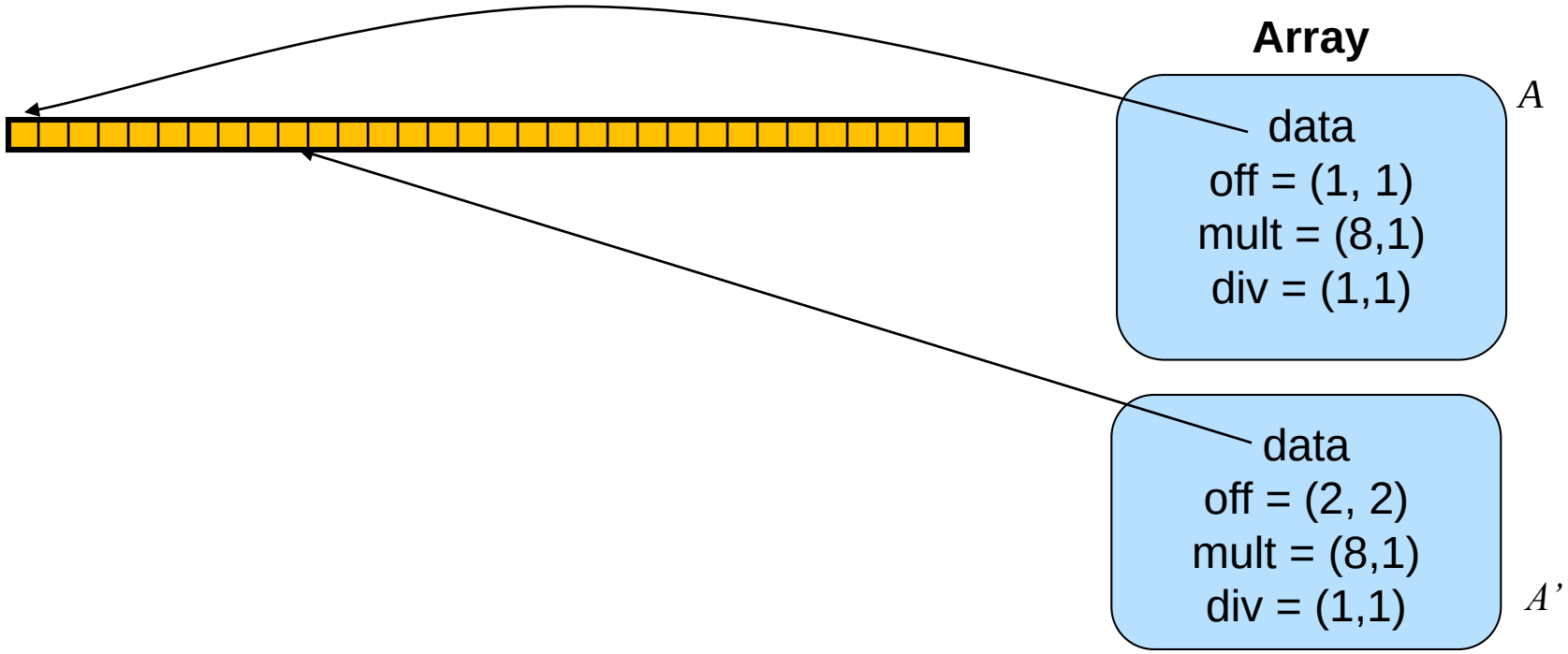
```
…A.reindex({0..m-1, 2..2*n + 2}); // use alternate indices to refer to A
```

*Historically, domain map descriptors have implemented these operations as part of their standard interface*

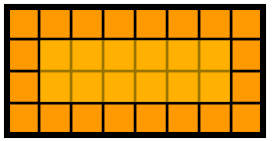
# Problems with this Approach

- **Implementing these three views can be nontrivial**
  - typically the last things domain map authors write, if they ever do
  - most distributions have had restrictions in practice
    - e.g., “I can’t handle strided slices”
    - compile- or execution-time errors when encountered
  - current implementations typically use “closed form”
    - store result of view operations using the same descriptor format

# Closed-form Slice Descriptor




```
const MyRMO = new dmap(new RMO(here.numCores, parStrategy.rows));

const D = {1..m, 1..n} dmapped MyRMO,
      Inner = D[2..m-1, 2..n-1];

var A: [D] real;
…A[Inner]…
```



# Challenges with Closed-Form Approach

- **Representations aren't always well-suited for closed-form**
  - e.g., BlockCyclic distributions store a packed set of array blocks per locale

```
const D = {1..n, 1..n} dmapped BlockCyclic(…);

var A: [D] real;
```

  - That representation isn't well-suited for arbitrary slices:

```
…A[3.. by 7, i]…
```

    - how to store the interference pattern between stride 7 and *numLocales*?
    - how to represent this 1D slice as a packed set of 2D blocks?

# Array Views: This Effort

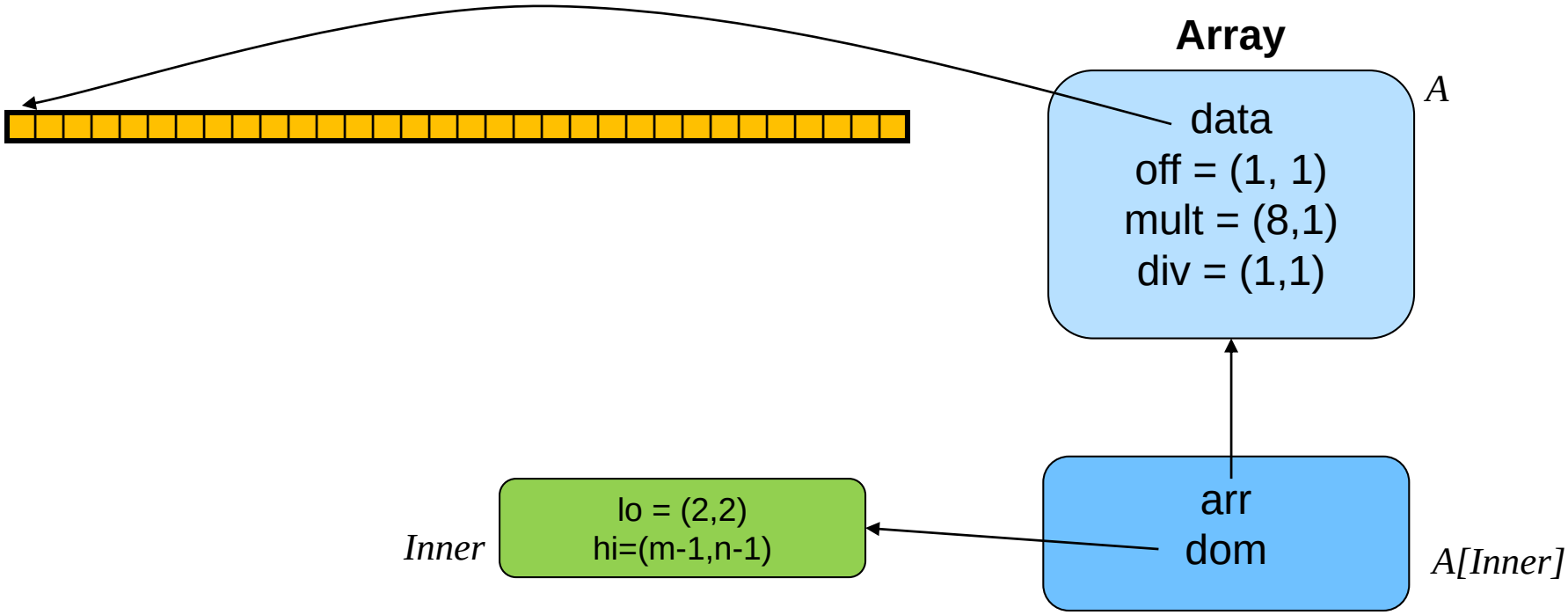
## This Effort:

- stop expecting domain maps to implement array view operations
- instead, introduce dedicated domain maps for these operations
  - have them implement the complete domain map interface
  - typically by forwarding to original descriptors

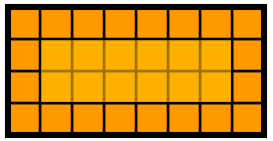
- e.g., for A[Inner]:
  - stop creating a new descriptor of A's type
  - instead, create a descriptor that
    - represents a slice expression
    - refers to the descriptors for 'A' and 'Inner'

- the following slides sketch the descriptors used for each operation
  - for each, the path that an access to that view takes is illustrated

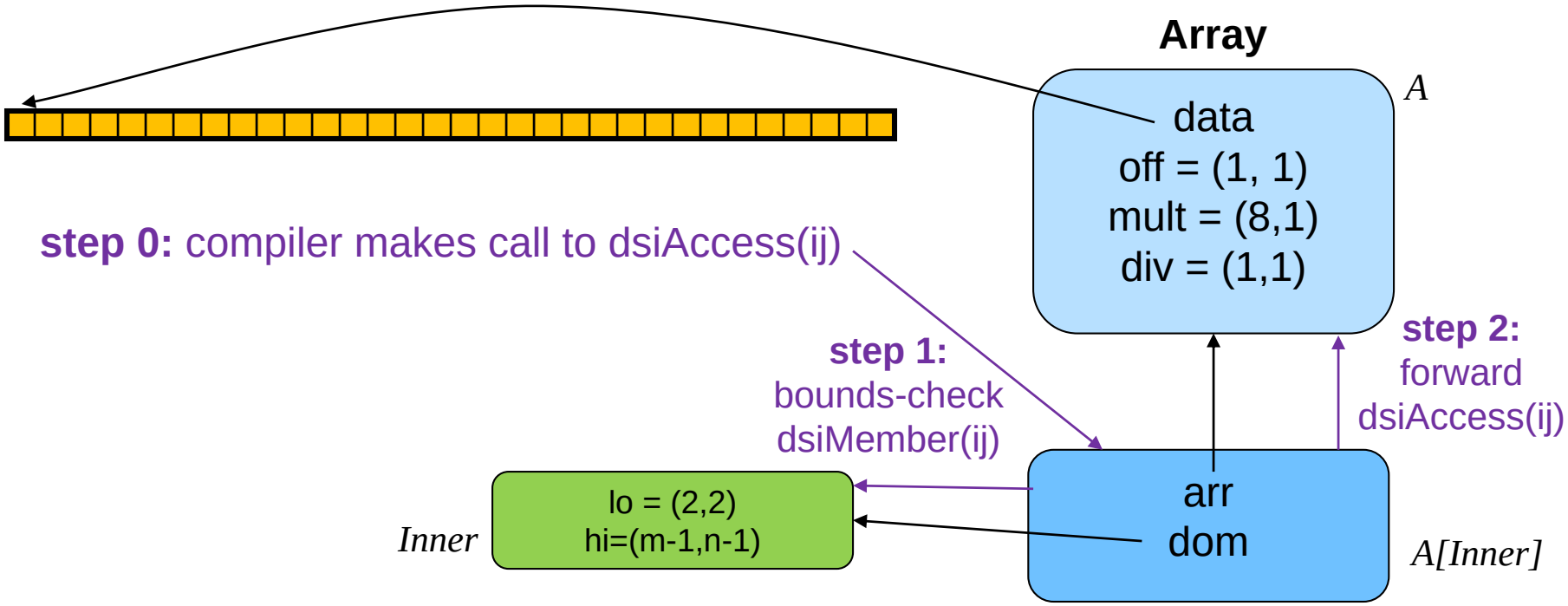
# Array View Slice Descriptor



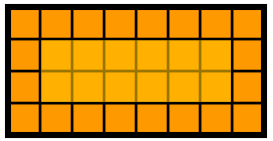

```
const D = {1..m, 1..n},
      Inner = D[2..m-1, 2..n-1];

var A: [D] real;
…A[Inner]…
```



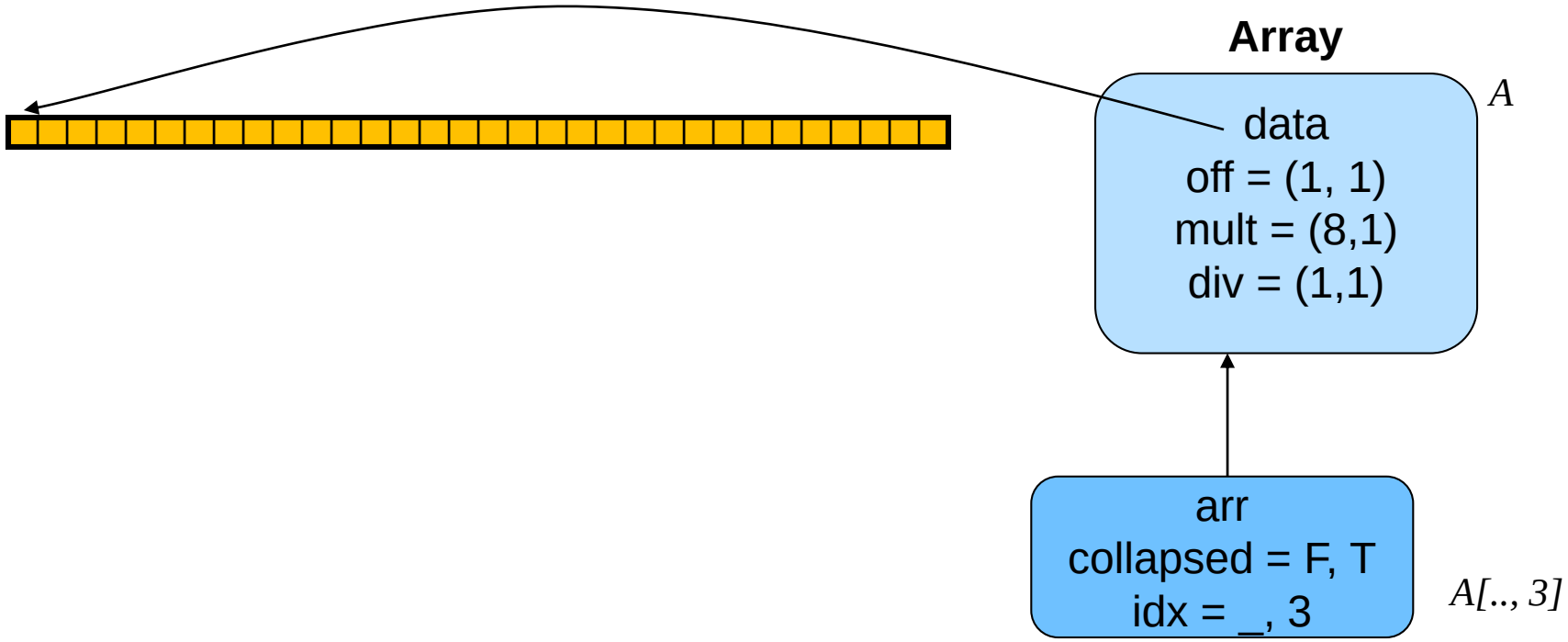
# Array View Slice Descriptor



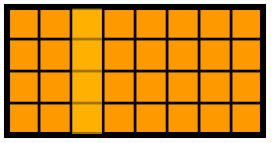

```
const D = {1..m, 1..n},
      Inner = D[2..m-1, 2..n-1];

var A: [D] real;
…A[Inner][ij]…
```



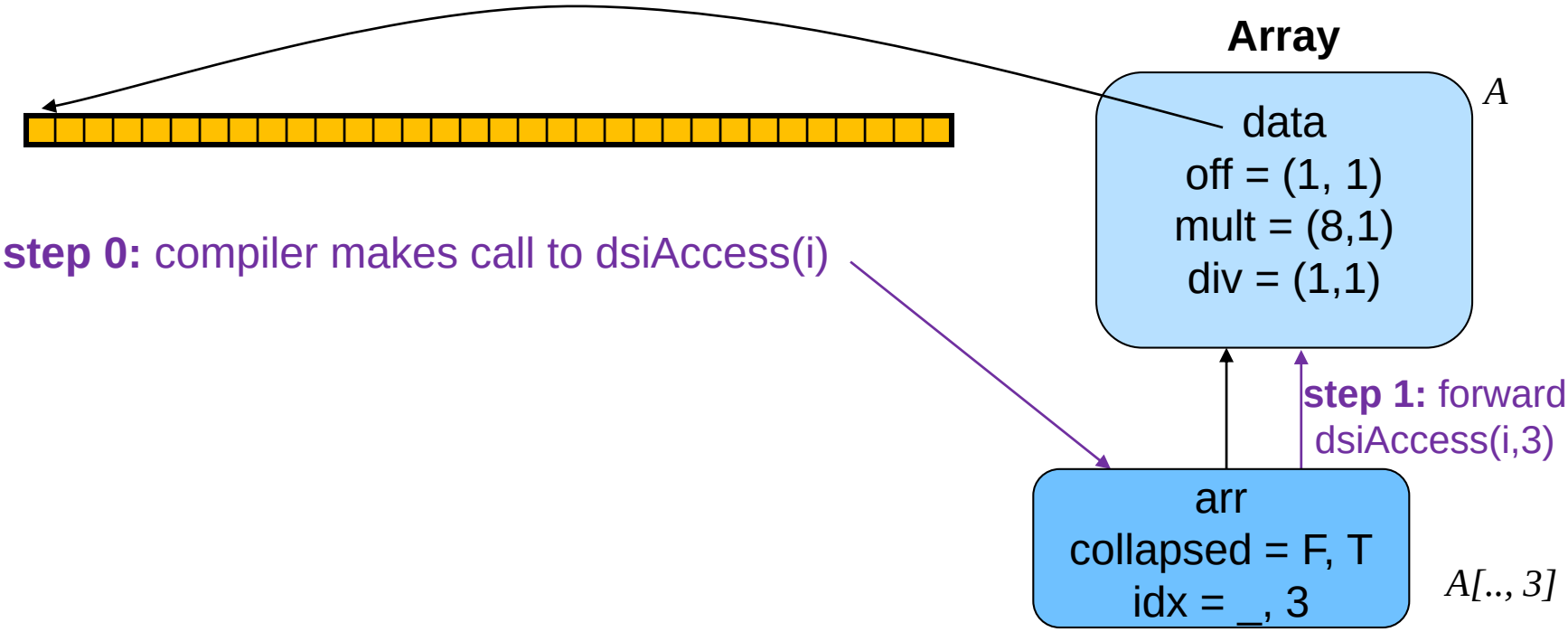
# Array View Rank-Change Descriptor



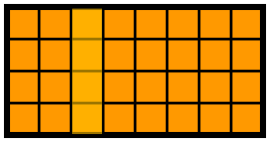

```
const D = {1..m, 1..n},
      Inner = D[2..m-1, 2..n-1];

var A: [D] real;
…A[.., 3]…
```



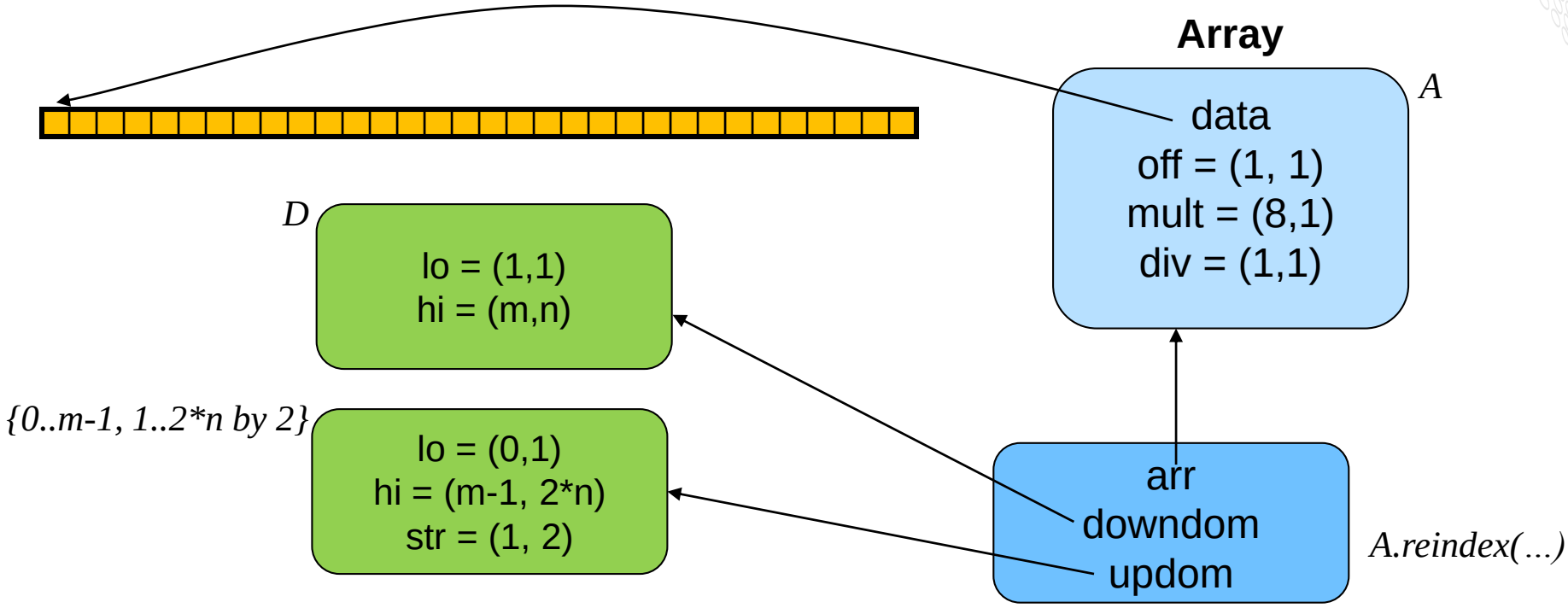
# Array View Rank-Change Descriptor



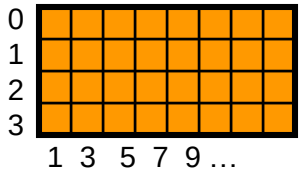

```
const D = {1..m, 1..n},
      Inner = D[2..m-1, 2..n-1];

var A: [D] real;
…A[.., 3][i]…
```



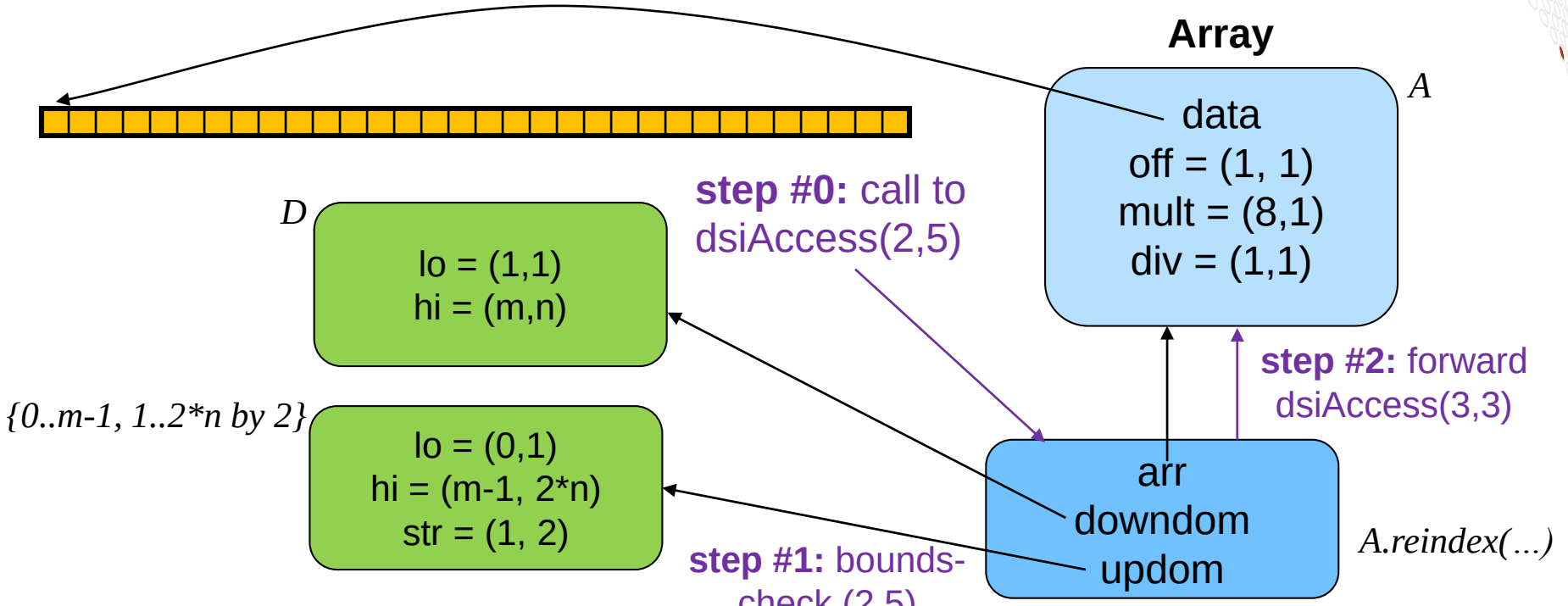
# Array View Reindex Descriptor



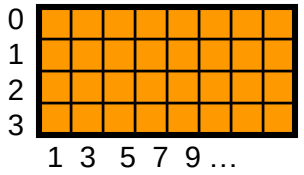

```
const D = {1..m, 1..n};

var A: [D] real;
…A.reindex({0..m-1, 1..2*n by 2})…
```



# Array View Reindex Descriptor




```
const D = {1..m, 1..n};

var A: [D] real;
…A.reindex({0..m-1, 1..2*n by 2})[2,5]…
```



# Domain / Domain Map Views

- **Rank-change and Reindex raise additional challenges**
  - Motivating example:

```
const D = {1..n, 1..n} dmapped Block(…);
var A: [D] real;   // A is a block-distributed 2D domain

foo(A[.., i]);     // pass a 1D slice of A to foo()

proc foo(X: [?Di]) {
  var B: [Di] real;   // Di must be a 1D domain representing D's ith column
  const S = {3..5} dmapped Di.dist; // Di.dist implies a domain map
}                     // that can create new 1D domains aligned with A's ith column
// yet, the original 2D domain and distribution are ill-equipped to support these ops
```

  - Reindex operations have similar challenges
  - Historically, domain map descriptors have had to support these ops
    - represented additional complexity for the author, often unimplemented
- **Solution:** Create views for domains and domain maps as well

# Array Views: Impact

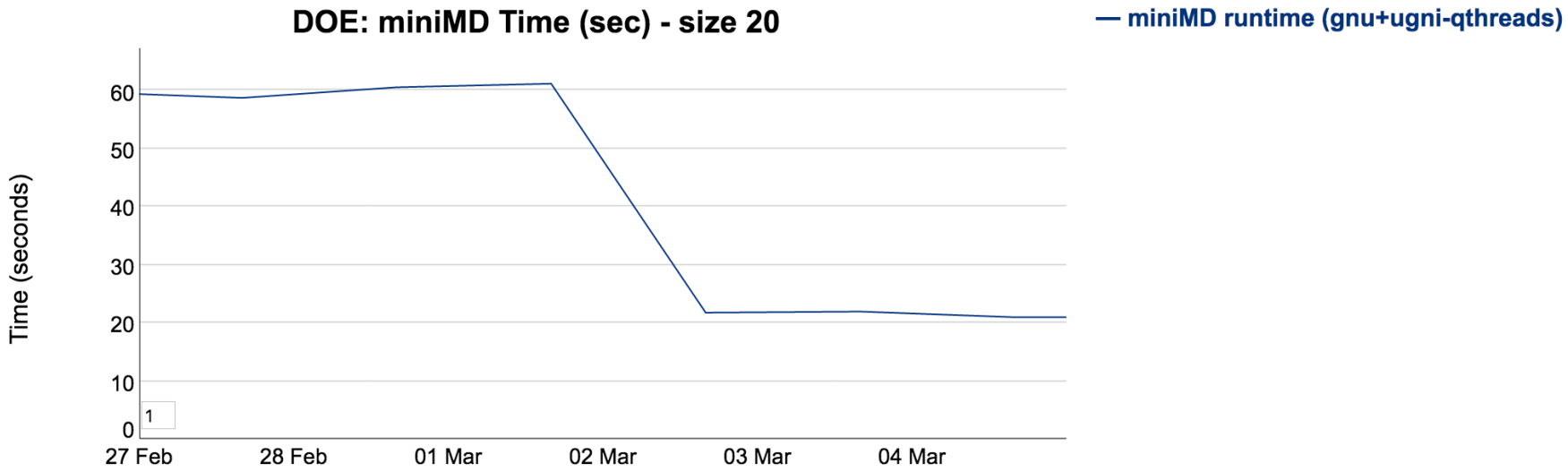
- **Simplified domain map standard interface**
  - eliminated the need for these previously required routines:

```
[array].dsiSlice()
[array].dsiRankChange()
[array].dsiReindex()
[dom].dsiBuildRectangularDom()
[dist].dsiCreateRankChangeDist()
[dist].dsiCreateReindexDist()
```

  - doing so eliminated a lot of complex code
  - as well as several previously unsupported cases:
    - strided reindexing of cyclic arrays
    - rank-change slicing of block-cyclic arrays
    - reindexing of block-cyclic arrays
    - ability to mix stridable- and non- in slicing Dimensional arrays
    - slices of block-cyclic arrays failed to bounds-check
    - …

# Array Views: Impact

- **Improved performance / reduced communication**
  - several tests that count communications for array operations improved
  - saw performance improvements, particularly for remote slice idioms
    - e.g., MiniMD uses remote slices to transfer boundary conditions
    - this graph illustrates resulting improvement for 16 compute nodes:

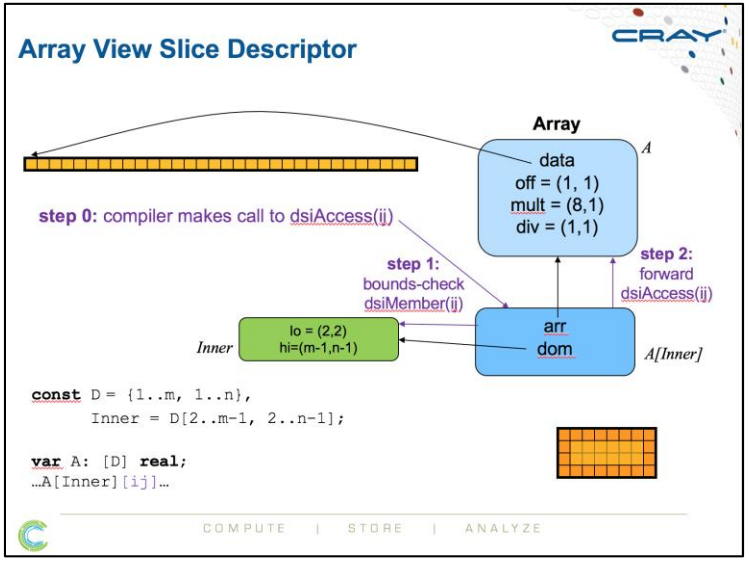


# Array Views: Impact

- **Enabled a more precise array indexing optimization**
  - Chapel 1.14 optimized array accesses using compiler analysis
  - Analysis known to be unstable w.r.t. unrelated arrays in the program
    - see “Array Indexing Optimization” slides in 1.14 release notes
  - Array views permit the optimization to be done in module code
    - simpler
    - more precise—no longer unstable w.r.t. unrelated arrays in the program
- **Creates distinct types for array view operations**
  - Arrays that “own” storage versus “alias” storage now more distinct
  - Improves ability to reason about and optimize such cases in compiler
- **Retired need for ‘=>’ in constructor context**
  - (see following section)

# Array Views: Impact

- **Downside: array view expressions now involve indirection**
  - in closed-form, array views were as fast as arrays
    - (when the operation was supported and worked…)
  - now, routines tend to be forwarded to the original array
    - believe this is the right trade-off
    - but important to be aware of




  - optimizations are possible
    - can squash stacked views
      - e.g., a slice of a slice of A can be stored as a slice of A
    - have already optimized common cases
      - e.g., we've manually optimized array views for default arrays

# Array Views: Next Steps

- **Though array views are implemented, some work remains**
  - **case 1:** loss of locality for domains/distributions of reindexed arrays:

```
const D = {1..n} dmapped Block(…);
var A: [D] real;
foo(A.reindex({0..n-1}));
proc foo(X: [?D]) {
  forall i in D do
    writeln(here.id);   // will always print 0 as D fails to preserve locality
}
```

  - **status:** reindex domain/dist. views drafted but not yet on master

# Array Views: Status

- **Though array views are implemented, some work remains**
  - **case 2:** can no longer pass array views to default constructors:

```
class C {
  var X: [1..3] real;
}
var A: [1..100] real;
var myC = new C(A[1..3]);
```

  - **reason:** compiler-generated constructor is too strict about types
    - slice-of-array no longer of identical type as array
  - **status:** planning to address with compiler-generated initializers

# Array Views: Status

- **Though array views are implemented, some work remains**
  - **case 3:** formal type queries + assertions changed in behavior:

```
proc foo(x: ?t, y: t) {…}

var A: [1..3] real, B: [1..10] real;
// in 1.14, all of the following worked:
foo(A, B[1..3]);          // no longer works in 1.15
foo(A[1..3], B[1..3]);    // no longer works in 1.15
foo(B[1..3], A);          // still works in 1.15
```

  - **reason:**
    - slices and arrays had same type in closed form, now they don't
    - code that implements '?t' inserts a copy
      - copies of slices result in deep copies ⇒ seems to have the type of a new array
      - so, all 3 cases want arg 2 to be an array rather than a slice
  - **status:** bug; needs to be addressed
    - probably by changing ?t implementation

# Deprecating Array Alias Operators '=>'

# Deprecating Array Alias Operators (=>)

- **Array alias operators were supported in two contexts:**
  - constructor calls
  - declarations
- **Chapel 1.15 deprecated support for both of these cases**
  - both represented special cases
  - neither was necessary any longer
- **We'll consider each case separately in the following slides**

# Deprecating Array Aliases in Constructor Calls

# => in Constructors: Background

- **Distributions like 'Block' store a domain + array per locale**
  - use a default local domain/array for these variables

```
class LocBlockArr {                    // class representing this locale's local block
  type eltType;                        // element type of the array
  var locDom: LocBlockDom;             // pointer to this locale's local domain
  var myLocElts: [locDom.myLocInds] eltType;     // this locale's block
}
```

- **For closed-form array views, this presented a challenge:**
  - consider the following slice of a block-distributed array:

```
const D = {1..n, 1..n} dmapped Block(…);
var A: [D] real;
…A[2..n-1, ..]…
```

  - since slices alias their original arrays…

    …the slicing expression's 'myLocElts' must alias A's 'mylocElts'
    - but it's an array field—how can we make it alias another array?

# => in Constructors: Background

- **Historical approach:**
  - added support for using the array alias operator in constructor calls
    - given:

```
class LocBlockArr {                          // class representing this locale's local block
  type eltType;                              // element type of the array
  var locDom: LocBlockDom;                   // pointer to this locale's local domain
  var myLocElts: [locDom.myLocInds] eltType; // this locale's block
}
```

    - the following calls would make the 'myLocElts' field alias an existing array:

```
var B: [1..10] real;
var C = new LocBlockArr(myLocElts=>B);
var myAslice = new LocBlockArr(myLocElts=>A.locArr.myLocElts);
```

- **Net result:**
  - supported closed form for such distributions
  - added a lot of complexity to the compiler
  - raised barriers to adding new generic fields to local arrays

# => in Constructors: This Effort

- **Array views no longer rely on closed form representations**
  - i.e., aliasing is now done by aliasing 'LocBlockArr', not 'myLocElts'

```
class LocBlockArr {                    // class representing this locale's local block
  type eltType;                        // element type of the array
  var locDom: LocBlockDom;             // pointer to this locale's local domain
  var myLocElts: [locDom.myLocInds] eltType;     // this locale's block
}
```

- **Thus, supporting '=>' in constructors is no longer needed**
  - Permits us to remove the related complexity from the compiler
    - and to remove a special case in the language
  - Makes it easier to add new generic fields to local arrays

# => in Constructors: Status and Next Steps

## Status:

- deprecated '=>' in constructors for 1.15
- plan to remove support for 1.16

## Next Steps:

- Do we want to support 'ref' fields in classes/records?
  - or perhaps fields that can switch between 'ref' and 'var' via a param?
- Generalizes the "refer, don't store" capability that this provided
  - yet in a way that is not array-specific
- No immediate need for this capability, but seems it could have utility

# Deprecating Array Aliases in Declarations

# => in Declarations: Background and Effort

## Background:

- Chapel has included a special array alias initializer

```
var A: [1..10] int;
var B => A;              // B refers to A's elements
var C: [0..9] => A;      // C refers to A's elements using indices 0..9
B[3] = 42;               // Now A[3] == 42
C[0] = 10;               // Now A[1] == 10
```

- Predated support for 'ref' declarations
  - Has felt increasingly like a redundant special-case since 'ref' was added
- Problematic in a few ways:
  - Fragile due to strict requirements on array types
  - Somewhat of a unique corner case in the language design

- **This Effort: Deprecated '=>' in favor of using 'ref' instead**

```
ref B = A;
ref C = A.reindex({0..9});
```

# => in Declarations: Impact and Next Steps

**Impact:** Language is simpler

- Removed an array-only language feature
- Less special-case code in compiler

**Next Steps:** Finalize other array alias questions

- How does one return an array alias without a deep copy?

```
var A: [1..10] real;
proc foo() {
  return A[1..3];   // copies out by default due to new array return semantics
}
```

  - But what if the user wants to return an alias to A?

```
proc foo() ref {   // one proposal: use 'ref' for such cases, as with full arrays
  return A[1..3];
}
```

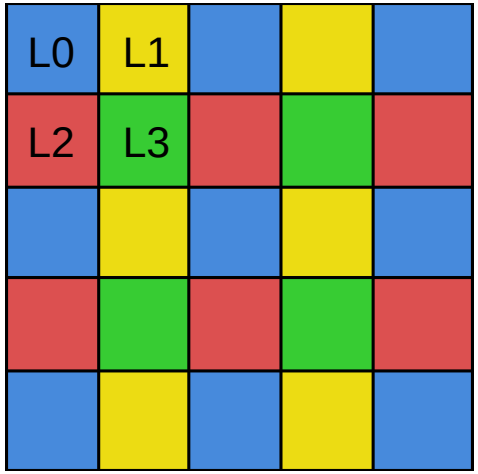
# BlockCyclic Improvements

# BlockCyclic: Background

- **BlockCyclic is a standard Chapel distribution**
  - Deals out blocks of indices in a round-robin fashion

- **Consider the following BlockCyclic domain:**

```
const D = {1..20, 1..20};
const Space = D dmapped BlockCyclic(startIdx=(1,1),
                                    blocksize=(4,4));
```




Distribution over 4 locales

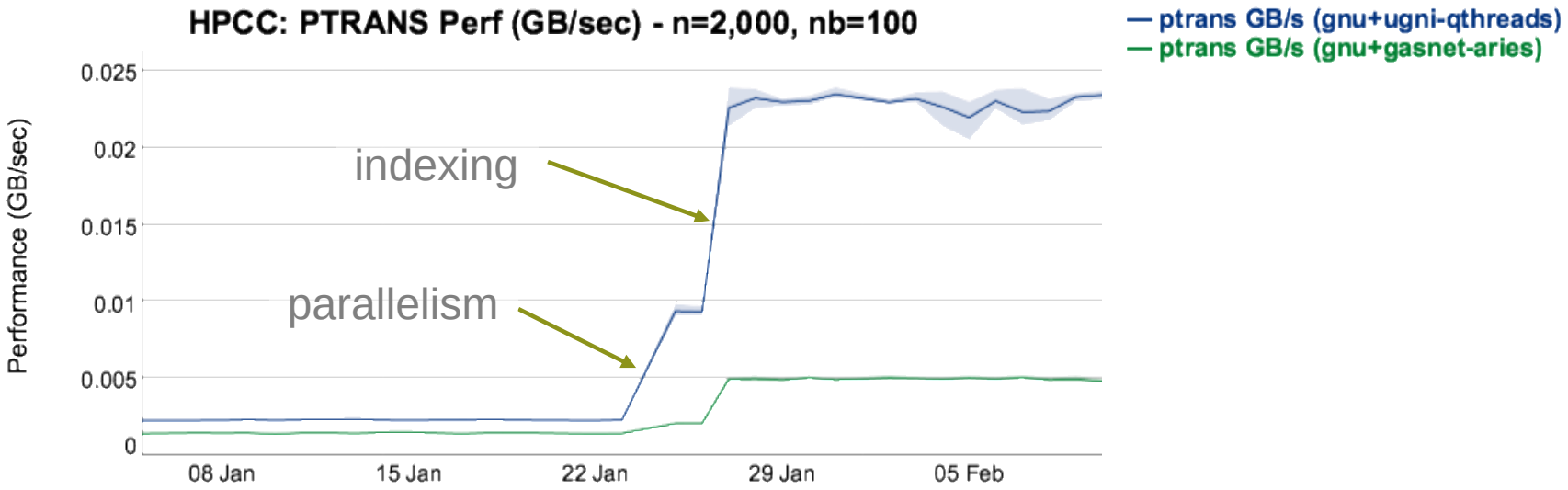
} 4x4 blocks of indices

# BlockCyclic: This Effort

- **Implemented intra-locale parallelism**
  - In 1.14, BlockCyclic only used one task per locale
- **Optimized indexing performance**
  - Use tuples instead of arrays for short-lived elements
  - Eliminated some incorrect multipliers
    - Fixed an out-of-bounds bug in the process

# BlockCyclic: Impact

- **10x improvement for HPCC PTRANS**
  - 16 nodes on Cray XC



# BlockCyclic: Status and Next Steps

## Status:

- BlockCyclic is faster and more correct in 1.15

## Next Steps:

- Continue improving indexing performance
- Improve privatization/caching
- Investigate wide-pointer overhead
- Investigate parallelism performance
  - Improvement is smaller than expected

# Array/Domain Shape Methods

# Shape Method

## Background:

- Array/domain shapes are useful in many contexts (e.g., linear algebra)

```
var A: [1..10, 3..30 by 10] real; // want a way of getting (10, 3) from A
```

- Getting array & domain shapes in a rank-neutral way was verbose:

```
var shape: A.rank*(A.dim(1).idxType);
for (i, r) in zip(1..shape.size, A.dims()) do
  shape(i) = r.size;
```

## This Effort:

- Add array.shape and domain.shape methods
  - Works on all flavors of domains and arrays

## Impact:

- Getting array/domain shapes has never been easier

```
const shape = A.shape;
```

# Other Array / Domain Map Improvements

# Other Array / Domain Map Improvements

- **For default arrays:**
  - added support for targetLocales() queries
  - fixed bugs in ‘sublocale’ queries based on identity of calling locale
- **Made .count() on arrays parallel by default**
- **Added comparator arguments to .sorted() on domains**
- **Reduced race conditions for associative-as-set operations**
- **Improved default hash functions for associative domains**

# Legal Disclaimer

*Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.*

*Cray Inc. may make changes to specifications and product descriptions at any time, without notice.*

*All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.*

*Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.*

*Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.*

*Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.*

*The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.*


CRAY

THE SUPERCOMPUTER COMPANY